Universiteit Gent
Faculteit Ingenieurswetenschappen
Vakgroep Informatietechnologie

Dynamische planning van rekenintensieve toepassingen in gedistribueerde computationele omgevingen

Dynamic Scheduling of Computationally Intensive Applications in Distributed Computing Environments

---

Maria Chtepen

Proefschrift tot het bekomen van de graad van
Doctor in de Ingenieurswetenschappen:
Computerwetenschappen
Academiejaar 2010-2011

Universiteit Gent
Faculteit Ingenieurswetenschappen
Vakgroep Informatietechnologie

Promotoren: Prof. Dr. Ir. Bart Dhoedt
Prof. Dr. Ir. Peter Vanrolleghem

Universiteit Gent
Faculteit Ingenieurswetenschappen

Vakgroep Informatietechnologie
Gaston Crommenlaan 8 bus 201, B-9050 Gent, België

Tel.: +32-9-331.49.00
Fax.: +32-9-331.48.99

Dit werk kwam tot stand in het kader van een
specialisatiebeurs van het IWT-Vlaanderen
(Instituut voor de aanmoediging van Innovatie door
Wetenschap en Technologie in Vlaanderen)

Proefschrift tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen:
Computerwetenschappen
Academiejaar 2010-2011

Dankwoord

Ik kom van ver, letterlijk en figuurlijk... Na jaren van studie, integratie in de Vlaamse samenleving en een zoektocht naar mijn eigen weg, ben ik op een punt gekomen waarop je kan zeggen dat alle inspanningen niet voor niets zijn geweest. Ik heb nu alles in mijn leven waarover ik vroeger kon dromen, en meer... Dit doctoraat is een kers op de taart die ik te danken heb aan een aantal ongelofelijke mensen die ik door de jaren heen heb ontmoet en die me zijn blijven motiveren en steunen om vooruit te gaan. In eerste plaats spreek ik over mijn man Filip die me gemaakt heeft tot wie ik nu ben. Ongeveer alles wat een doorsnee Belg moet kennen en kunnen heb ik van hem geleerd: zwemmen, auto rijden, skiën, kamperen, genieten van reizen en muziek, Engels en Nederlands spreken, mooi schrijven (al zeg ik het zelf), programmeren... Filip, ik kan niet in woorden beschrijven hoe dankbaar ik je ben. Natuurlijk kan ik het hier niet bij laten zonder mijn twee lieve schatjes te vernoemen, Helena en Elise, die mijn leven hebben gevuld met liefde, trots en blijdschap.

Dit dankwoord sluit voor mij niet enkel de 5 jaar van onderzoek bij IBCN af maar ook mijn eerste 10 jaar in België. Daarom hoop ik dat ik word vergeven voor het afwijken van de standaard procedure en voor het verder vermelden van mensen, die een belangrijke rol hebben gespeeld in die periode van mijn leven, in chronologische volgorde...

In de eerste plaats, zou ik hier nooit geweest zijn zonder mijn grootouders Aili en Alexander, die me met veel liefde en zorg hebben opgevoed, en zonder mijn moeder die mijn beste vriendin geweest is door de moeilijke jaren heen. Verder dank aan mijn goede vrienden Paula en Bernard. Jullie waren aanvankelijk totale onbekenden die voor mij meer hebben gedaan dan een eigen familie zou doen. Uiteraard ben ik ook bijzonder dankbaar aan mijn schoonfamilie, Nicole, Hubert, Katrien, Peter, Thomas en Mathias, om dat raar Russisch meisje, dat ik in het begin was, te aanvaarden en een hechte familie voor mij te worden.

Wat betreft mijn professionele ontwikkeling, vermeld ik met veel plezier professoren Piet Demeester en Peter Vanrolleghem die een kader hebben geschetst voor zowel mijn licentiaatsthesis als doctoraat, door een samenwerking op te starten tussen hun vakgroepen, INTEC en BIOMATH. Ik dank Peter ook in het bijzonder voor zijn input en wetenschappelijk inzicht dat hij met mij deelde als co-promotor van dit doctoraat. Een andere persoon die uiteraard zeer substantieel heeft bijgedragen tot het uitstippelen, sturen en opvolgen van dit onderzoek is promotor prof. Bart Dhoedt. Verder dank ik iedereen die mijn werk financieel, technisch of mentaal heeft ondersteund: de vakgroep INTEC en het Instituut voor de aan-

moediging van Innovatie door Wetenschap en Technologie in Vlaanderen (IWT) voor de beurzen die ik kon genieten gedurende 5 jaar; het technisch en administratief personeel van IBCN en IBBT voor hun stiptheid en behulpzaamheid; mijn ex-collega's Jens, Jos, Stefan, Nicolas, Koen, Jan, Dieter (x2), Bas, Raf, Matthias, Kevin, Kristof, Tim, Frederic, Lien, Johannes, Bart (x2), Sofie (x3), Bruno (x2), Jonathan, Peter, Pieter (x2), Eric, Stijn (x2) en Elisabeth voor de aangename tijd bij IBCN.

Ik wil ook graag een aantal toffe madammen vermelden die sinds een paar jaar mijn avondleven helemaal hebben veranderd... Chantal, Veerle, Mieke, Elke, Sylvie en Katleen: dankzij jullie heb ik ontdekt dat een getrouwde vrouw met 2 kinderen nog iets anders op een avond kan doen dan kuisen of TV kijken, zoals zwemmen, squashen, een kookcursus volgen, naar de bioscoop gaan of simpelweg gezellig iets gaan drinken.

Afsluitend wil ik het management van BelV en mijn superieuren Pieter De Gelder en Marc Dubois bedanken voor de mogelijkheid om dit doctoraat af te ronden door gebruik te maken van thuiswerken en een flexibele uurregeling. In het bijzonder dank ik ook Marcel Maris om te geloven in mijn capaciteiten en voor zijn onmisbare adviezen.

Indien nog niet vernoemd, dank ik ook jou, lezer van dit boek, voor je interesse in dit werk!

Sint-Martens-Latem, oktober 2010
Maria Chtepen

Samenvatting

Dit proefschrift behandelt de kwestie van de efficiënte uitvoering van computationeel intensieve toepassingen binnen gedistribueerde omgevingen, zoals clusters, peer-to-peer netwerken en grids. Meer in het bijzonder, onderzoeken we hoe runtime informatie over de status van het systeem kan worden opgenomen in het job scheduling proces. Typisch voor grote gedistribueerde systemen is dat hun rekencapaciteit en beschikbaarheid sterk kan variëren in functie van de tijd. Bovendien zijn hun resources vaak gedeeld door verschillende gebruikersgroepen, wat resulteert in een hoge diversiteit en uiteenlopende complexiteit van de uitgevoerde applicaties (jobs). Deze kenmerken suggereren dat schedulers die rekening houden met dynamische systeem- en job-veranderingen, een belangrijke meerwaarde kunnen bieden in vergelijking met klassieke statische oplossingen.

Het proces van de gedistribueerde uitvoering brengt verschillende uitdagingen met zich mee, dewelke zich situeren op zowel organisatorisch als op soft- en hardware niveaus. In deze context richten we ons op twee belangrijke problemen, die een aanzienlijk effect hebben op de prestaties en de flexibiliteit van gedistribueerde omgevingen: fout-tolerantie en scheduling van toepassingen met afhankelijkheden (workflows). Verschillende dynamische checkpointing-, replicatie- en schedulingoplossingen worden voorgesteld die rekening houden met het dynamische karakter van gedistribueerde systemen en hun toepassingen, door het herevalueren van eerder genomen beslissingen at run-time.

Om het gedrag van de voorgestelde oplossingen te bestuderen binnen dynamische grid-omgevingen werd gebruik gemaakt van een discrete-event simulator, genaamd DSiDE. DSiDE werd ontwikkeld in het kader van dit onderzoek en vormt een flexibel en portabel framework voor modellering en simulatie van gedistribueerde computationele omgevingen. Tot dusver bevat DSiDE voornamelijk ingebouwde grid-componenten, die echter gemakkelijk kunnen worden uitgebreid met modellen voor andere soorten van gedistribueerde systemen, zoals P2P-netwerken en clouds. De belangrijkste voordelen van DSiDE, in vergelijking met andere bestaande algemene en grid-specifieke simulatoren, zijn uitbreidbaarheid, genericiteit, korte simulatietijden en de mogelijkheid tot eenvoudig modelleren en simuleren van het dynamisch gedrag van gedistribueerde systemen en applicaties. Het dynamisch gedrag ondersteund door DSiDE omvat onder andere wisselende belasting en beschikbaarheid van resources, variërende aankomstfrequenties van jobs, veranderende applicatie dynamiek, *etc*. In het algemeen bestaat de architectuur van de simulator uit drie afzonderlijke modules:

- **DGen**: grid-modellen en het dynamisch systeemgedrag worden gespecificeerd als input in de simulator met behulp van een XML-formaat. Eventpatronen uit de input file worden vertaald door DGen in een reeks van individuele events die kunnen worden geladen in de DExec module.
- **DExec**: is de kern van DSiDE, waar simulaties worden uitgevoerd door de sequentiële verwerking van geregistreerde events.
- **DMExec**: zorgt voor automatische uitvoering van een aantal voorgedefinieerde simulatie-experimenten of van hetzelfde experiment met verschillende seeds voor random number generators.

Het eerst behandelde onderwerp is fout-tolerantie van grid-systemen. We beschouwen onbetrouwbare grids, d.w.z grids waar resources onderworpen zijn aan het regelmatig falen en heropstarten, waar toepassingen van verschillende duur worden uitgevoerd. Het doel van deze studie is om een kwantitatieve indicatie te geven van het effect van het falen van resources op de grid-prestaties en om het gebruik van fout-tolerante technieken in deze dynamische omgevingen te rechtvaardigen. De resultaten hebben aangetoond dat een aanzienlijke vermindering van grid-performantie wordt bereikt in onstabiele systemen voor jobs met lange uitvoeringstijden. Een andere conclusie is dat de frequentie van het falen van een resource een groter effect op de grid prestaties heeft dan de tijd die nodig is om een resource te herstellen. Als gevolg van deze studie werden verschillende dynamische fout-tolerantie algoritmen voorgesteld. De algoritmen zijn gebaseerd op bekende replicatie- en checkpointing-technieken. Het belangrijkste probleem met deze technieken is de grote overhead geïntroduceerd wanneer het aantal replica's en het checkpointing-interval niet optimaal worden gekozen. Veel bestaand onderzoek op dit gebied is gewijd aan het analytisch bepalen van de waarden van beide parameters, gebaseerd op kennis van de toepassing en de gedistribueerde omgevingen. Helaas baseren de daaruit voortvloeiende oplossingen zich vaak op veronderstellingen of onrealistische vereenvoudigingen. De algoritmes geïntroduceerd in dit werk anderzijds, wijzigen het aantal replica's en het checkpointing-interval dynamisch, respectievelijk op basis van run-time informatie over de belasting van het systeem en de geschiedenis van resource fouten. Simulatie-resultaten hebben aangetoond dat adaptieve replicatie de meest low-cost aanpak is in systemen met een lage en variabele belasting, terwijl in zwaar beladen omgevingen, de dynamische checkpointing-intervalselectie meestal resulteert in een aanzienlijke vermindering van run-time overhead, in vergelijking met periodieke checkpointing. De voordelen van beide technieken kunnen worden gecombineerd in een hybride aanpak die het best kan worden gebruikt als de systeemeigenschappen niet vooraf bekend zijn.

Een andere kwestie die wordt behandeld in dit onderzoek is de uitvoering van workflow-toepassingen met input-afhankelijkheden. Input-afhankelijkheden impliceren dat een taak binnen een job input genereert die vereist is voor de uitvoering van een andere taak. Dit soort afhankelijkheden introduceert beperkte runtime communicatie-overhead en levert een aanzienlijke prestatieverbetering bij decentrale uitvoering van parallelle taken. Daarom stellen wij een dynamisch algo-

ritme voor workflows voor dat gebaseerd is op de veronderstelling dat de taken binnen een job wisselende computationele complexiteit vertonen. Het algoritme werkt op jobs waarvoor de vooruitgang van de uitvoering kan worden opgevraagd at run-time. Op basis van de vooruitgang en op basis van de huidige resource-capaciteiten, worden de taken herscheduled zodanig dat de uitvoeringstijden van parallele taken met dezelfde afhankelijkheden in balans blijven. Het idee is dat een *afhankelijke taak* niet kan worden uitgevoerd totdat al de outputs van haar *antecedenten* gegenereerd zijn. Daarom is het wenselijk om aan antecedenten met korte uitvoeringstijden trage computationele resources toe te kennen en om de snelle resources voor te behouden voor taken die een snelle verwerking vereisen. De prestaties van het voorgestelde algoritme werden geëvalueerd met behulp van een workload model afgeleid van een real-world tool voor het modelleren en simuleren van milieusystemen, genaamd Tornado. De resultaten suggereren een workflow makespan vermindering van ongeveer 35 %. Toch is het belangrijk te vermelden dat de prestatie van het algoritme afhankelijk is van de kwaliteit van de voorspellingen van de taakuitvoeringstijden, vermits deze voorspellingen worden gebruikt voor evenwichtige (re)scheduling. Aanvankelijk werden de uitvoeringstijden voorspeld door extrapolatie van de laatste meting van de taakvooruitgang. Helaas, deze eenvoudige methode is gevoelig voor interne dynamiek van toepassingen en voor variaties in resource-belasting, wat leidt tot overijverig en overhead-gevoelige taakmigraties binnen elke algoritme-iteratie. Om deze overhead te reduceren, werd een meer doeltreffende voorspellingsmethode ingevoerd. De methode is gebaseerd op niet-lineaire curve-fitting van de voorspellingen (gebouwd door extrapolatie) tegen een aantal vooraf gedefinieerde voorspellingsmodellen. Simulatie-experimenten met Tornado werkload suggereren tot 15 % prestatieverbetering in geval van curve-fitting.

Summary

In this dissertation we address the issue of efficient execution of computationally intensive applications within distributed environments, such as clusters, peer-to-peer networks, grids. More specifically, we investigate how to incorporate runtime information in the job scheduling process. Typical for large distributed systems is that their computational capacity and availability can strongly vary over time. Furthermore, their resources are often shared among different user groups, resulting in high diversity and varying complexity of application (jobs) runs.

There are different challenges related to the process of distribution execution, which are situated at the organizational as well as at software and hardware levels. In this context, we focus on two important problems, which have significant impact on the performance and flexibility of distributed environments: fault-tolerance and scheduling of applications with dependencies (workflows). Several dynamic checkpointing, replication and scheduling solutions are proposed that take into account the dynamic nature of distributed resources and applications, by reconsidering previously taken decisions at run-time.

To study the behaviour of grid environments in dynamic scenarios, we have developed a discrete-event simulator, called DSiDE. DSiDE forms a flexible and portable framework for modeling and simulation of distributed computing environments. Thus far, it mainly contains a set of built-in grid components, which, however, can easily be extended with components for other types of distributed systems, such as P2P networks and clouds. The main advantages of DSiDE, compared to other existing general and grid-specific simulation frameworks, are its extensibility, genericity, relatively short simulation times and ability to easily model dynamic system and application behaviour. Dynamic behaviour supported by DSiDE includes varying resource load and availability, varying job arrival frequency, changing application dynamics, *etc*. In general, the simulator is composed of three separate modules:

- **DGen**: grid models and dynamic behaviour modeling event distributions are provided as input into the simulator using an XML-based format. Recurrent events from the input file are translated by DGen into a set of individual events that can be loaded into the DExec module.
- **DExec**: is the kernel of the simulator, where simulations are run by processing all registered events one after the other.
- **DMExec**: allows for automatic execution of either several predefined simulation experiments or of the same experiment with different seeds for ran-

dom number generators.

The first issue addressed is fault-tolerance in grid systems. We consider unreliable grids, *i.e.* grids where resources are subject to failure and restart, where applications of different duration are executed. The aim of this study is to give a quantitative indication of the effect of resource failure on grid performance and to justify the use of fault-tolerant techniques in these dynamic environments. The results have shown considerable performance degradation of jobs with long execution times in unstable systems. Also, they suggest that the frequency of resource failure has a larger effect on grid performance than the time it takes a resource to restore. As a consequence of this study, several dynamic fault-tolerance algorithms were proposed. The algorithms are based on well-known job replication and checkpointing techniques. The main issue with these techniques is the large overhead introduced when the number of replicas and the checkpointing intervals are chosen inappropriately. Many existing research efforts in this area are dedicated to determining the values of both parameters analytically, based on knowledge of the application and distributed environments at hand. Unfortunately, the resulting solutions are often based on unrealistically simplified assumptions or limited to a certain type of applications. Therefore, the algorithms introduced modify the number of replicas and the checkpointing intervals dynamically, based on run-time information on system load, and the history of resource failures respectively. Simulation results have shown that adaptive job replication is the most low-cost approach in systems with low and variable load, while in heavily loaded environments, checkpointing with dynamic interval can significantly reduce run-time overhead, compared to periodic checkpointing. The advantages of both techniques can be combined in a hybrid approach that can best be utilized when system properties are not known in advance.

Another issue addressed in this research is scheduling of workflow applications with input dependencies. Input dependencies imply that a task within a job requires inputs generated by another task before it can proceed with its execution. This is a loosely-coupled type of dependencies with limited run-time communication overhead, which can gain significant performance improvement from distributed execution of parallel tasks. Therefore, we propose a dynamic algorithm for workflows that is based on the assumption that tasks within a job have varying computational complexity. The algorithm operates on applications for which execution progress can be monitored at run-time. Based on the monitored task progress and current resource capacity, tasks are (re)scheduled to resources to keep the execution times of parallel tasks with the same *dependents* in balance. The idea is that a *dependent* task cannot be executed until all its *parents* have generated the required outputs. Therefore, it is desirable to assign *parents* with low execution times to slow computational resources and keep fast resources for tasks requiring fast processing. The performance of the algorithm proposed was evaluated using a workload model derived from a real-world tool for modeling and virtual experimentation with environmental systems, called Tornado. The results suggested a makespan reduction of about 35% for the job as a whole. However, it is important

to mention that the performance of the algorithm depends on the quality of task execution time predictions, since these predictions are used to balance the execution time of running tasks. Initially, execution time predictions are constructed by extrapolation from the last task progress measurements. Unfortunately, this simple method is sensitive to internal application dynamics and variations in resource load, which leads to overzealous and overhead-prone task migrations within each scheduling iteration. To reduce this overhead, a more effective prediction method is introduced. The method is based on non-linear curve-fitting of predictions (constructed by extrapolation) against a number of predefined prediction evolution models. Simulation experiments with Tornado workload suggested up to 15% performance improvement in case the curve-fitting based prediction method is applied to the dynamic algorithm proposed.

Table of Contents

List of Figures

List of Tables

List of Symbols and Acronyms

Symbols

A	Amplitude of the execution time prediction evolution oscillations
A_{grid}	Grid computational availability
A_{CR}	Computational availability of resource CR
AR_J	Number of active replicas of job J
b	Weight factor determining speed of decrease/increase of oscillation amplitude over time
B	Network bandwidth
C	Checkpointing run-time overhead
CA	Number of active CPUs
CL	Lower boundary of active free CPUs for replication to take place
CR	Computational Resource instance
CR^{ref}	Reference Computational Resource with $MIPS_{CR^{ref}} = 1$ and $n_{CR^{ref}} = 1$
$CSize_J$	Checkpoint size of job/task J
E^{avg}	Average estimated execution time
E_J^{act}	Actual execution time of task J on theoretical reference machine
E_J^{CR}	Execution time of J on resource CR
E_J^{est}	Estimated execution time of task J on theoretical reference machine
$Edges$	Directed Acyclic Graph dependency flows
E_J^{ref}	Reference estimated execution time
F	Oscillation curve period
F_{CR}	Mean time between failures of resource CR
$G_{\#}$	Number of Computational Resource groups
γ	Execution time improvement percentage thanks to migration
I	Checkpointing interval
I_{GS}	Job scheduling interval

I_{IS}	System status collection interval of IS
I_J^{min}	Minimum checkpointing interval of job J
I_J^{opt}	Optimal checkpointing interval for job J
I_J^{CR}	Customized checkpointing interval for job J running on resource CR
In_J	Input data size
J	Computational Job or Task
k	Number of Running Parent Sets to be scheduled
l	Network link
l_b	Bottleneck link
$\lambda_r^{l_b}$	Bandwidth allocated to route r when $l_b \in r$ is the bottleneck link
L_l	Latency of link l
LF_{CR}	Last detected failure of resource CR
MF_{CR}	Mean failure interval of resource CR
$MIPS_{CR}$	Speed or capacity of resource CR
$MIPS_J^{CR}$	Share of CR speed/capacity allocated to job J
$MIPS_{CR}^{total}$	Total capacity of resource CR
$MIPS^{ex}$	Background load
$MIPS_S$	Capacity of site S
n	Number of Computational Resources in grid
N_{batch}	Number of simulations in batch
n_{CR}	Number of jobs/tasks currently scheduled to CR
n_{CR}^{max}	Maximum number of jobs/tasks allowed to run on CR
$Nodes$	Directed Acyclic Graph application tasks
NO_r^J	Network overhead of transferring task J over route r
N_{seed}	Number of different seeds for simulation runs
N^{total}	Total number of resources in grid
ω	Share of link bandwidth
P	Percentage of increase/decrease in Computational Resource load
P_J	Percentage of task J workload already completed
ϕ_r	Round-trip time of route r
ψ	Prediction evolution function
PS_J	Parent set of task J
r	Network route
r_1	Random weight factor for execution time prediction evolution models
r_2	Random weight factors for execution time prediction evolution models
r_3	White noise amplitude
R	Checkpoint recovery delay
Rep	Number of job replicas
Rep^{max}	Upper replica number boundary

Rep^{min}	Lower replica number boundary
RE_J^{CR}	Remaining execution time of job J on resource CR
$RE_{J,CR}^{est}$	Estimated remaining execution time of task J on resource CR
RE^{min}	Minimum remaining execution time required to proceed with rescheduling
$RS(t)$	*Ready* or *Parallel set* at timestamp t
t	Current simulated time
t_c	Current system time
$t_{CR,n}^f$	Timestamp of resource CR failure
$t_{CR,n}^r$	Timestamp of resource CR restart
θ	Maximum of resources within each group
t_i	Wall clock time expired between the beginning of job execution and the initial checkpoint
T_J	Currently expired wall clock execution time of task J
T^{sim}	Total wall clock job execution time
T^{total}	Total simulated job/task execution time
φ	Oscillation curve phase
$\mathcal{W}$	Directed Acyclic Graph instance

A

AEST	Absolute Earliest Start Time
AHLFET	Application Highest Level First with Estimated Times
AICPDP	Application Improved Critical Path using Descendant Prediction
ALST	Absolute Latest Start Time
Amazon EC2	Amazon Elastic Compute Cloud
AMCP	Application Modified Critical Path

B

BEgrid	Belgian compute/data grid infrastructure for research

C

CEC	Current Event Chain

CERN	European Organization for Nuclear Research
ClassAd	Classified Advertisement language
CLI	Commond-Line-Interface
CODINE	COmputing in Distributed Networked Environments
CPU	Central Processing Unit
CR	Computational Resource
CS	Checkpointing Server

D

DAG	Directed Acyclic Graph
DAGMan	Directed Acyclic Graph Manager
DLL	Dynamically Linked Library
DSiDE	Dynamic Scheduling in Distributed Environments

E

EGEE	Enabling Grids for E-sciencE
FJBCR	First Job Best Computational Resource

F

FCFS	First Come First Served
FEC	Future Event Chain
FOSS	Free and Open Source Software

G

GS	Grid Scheduler

H

HBM	Heartbeat Monitor Interface

HEP	High Energy Physics
HLFET	Highest Level First with Estimated Times
HPC	High-performance computing

I

IaaS	Infrustructure as a Service
ICPDP	Improved Critical Path using Descendant Prediction
IDE	Integrated Development Environment
ILP	Integer Linear Programming
IO	Input/Output
IS	Information Service

J

JDL	Job Description Languuage

L

LAN	Local Area Network
LCG	LHC Computing Grid
LHC	Large Hadron Collider
LSF	Load Sharing Facility

M

MCP	Modified Critical Path
MIPS	Million Instructions Per Second
MPI	Message Passing Interface

O

OpenPBS	Open Portable Batch System

OS	Operating System

P

P2P	Peer-to-Peer
PaaS	Platform as a Service
PBS	Portable Batch System
PC	Personal Computer

Q

QoS	Quality of Service

R

RR	Round Robin
RNG	Rundom Number Generator
RPS	Running Parent Set

S

S	Grid Site
SaaS	Software as a Service
SETI	Search for Extra-Terrestrial Intelligence
SGE	Sun Grid Engine
SLA	Service Level Agreement
SR	Storage Resources
SSE	Sum of Squared Errors

T

TCD	Trust, Cost and Deadline based schedule
TCP	Transmission Control Protocol

Torque	Terascale Open-Source Resource and QUEue Manager

U

UI	User Interface

V

VM	Virtual Machine

W

WAN	Wide Area Network
WQR	Workqueue with Replication
WWTP	Waste Water Treatment Plant

X

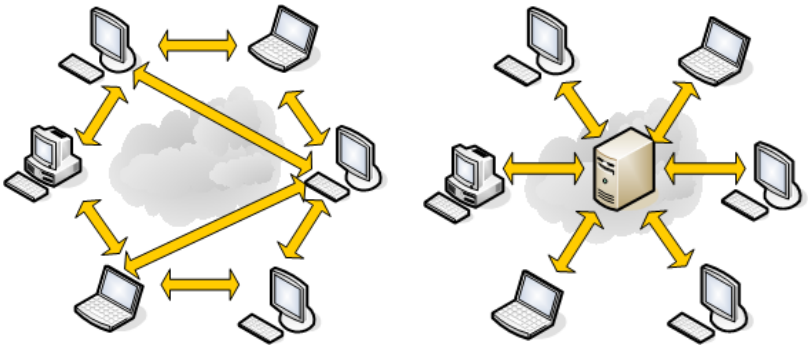
XML	eXtensible Markup Language

1 Introduction

This chapter provides a short introduction to distributed computing, its application areas and challenges. Focus is on optimization of distributed execution of computationally intensive applications by means of intelligent scheduling. Related work in this area is introduced, together with an outline of this dissertation.

1.1 Distributed Computing

By the term "distributed computing" we understand the execution of an application on a number of distributed computational resources connected by computer networks. Distributed computing differs from the well-known client/server paradigm by the granularity of the distribution process. In a client/server system an application is submitted integrally from a client to a remote server where it is sequentially processed. In distributed computing, typically, jobs are initially subdivided into independent or partially dependent tasks, each of which can be executed on a different resource. This process is called "gridification" and its main purpose is parallelisation of applications to achieve computational speed-up. Generally, only computationally intensive jobs can benefit from parallel execution, since gridification and task distribution impose significant overhead that can exceed run times of short applications. Therefore, distributed computing is usually applied in computationally complex domains, such as high-energy physics, pharmaceutical drug discovery, economics, weather forecasting, *etc*.



(a) (b)

Figure 1.1: An example of a Peer-to-Peer architecture:(a) a P2P system without central server; (b) a P2P system with central server.

1.1.1 Distributed Computing Technologies

The four most common technologies in the domain of distributed computing are: peer-to-peer computing, cluster computing, grid computing and cloud computing.

Peer-to-peer computing, often abbreviated to P2P, utilizes P2P networks (see Figure 1.1) for data sharing or execution of distributed applications. P2P networks are composed of distributed desktop machines or servers sharing a portion of their processing power, storage resources or network resources with other users typically within a Wide Area Network (WAN). In a P2P network each node is a supplier and a consumer of resources at the same time, in contrast to the client/server approach where each machine has its own predefined non-overlapping role. P2P networks are formed by *ad hoc* addition of nodes and their capacity strongly depends on the number of available nodes at each point in time. However, the weakness of P2P is at the same time its strength since the distributed and dynamic nature of this network provides for enhanced scalability and service robustness. As shown in Figure 1.1 we differentiate between two P2P architectures: peers that communicate through direct connections established between nodes that know each other; and peers that connect through a central server that is used for system bootstrap and indexing. A well-known P2P application is SETI@home (Search for Extra-Terrestrial Intelligence) [1]. The SETI project uses scientific methods to search for electromagnetic transmissions from civilizations on distant planets. These methods result in a large number of computationally demanding tasks that are executed on distributed community resources.

A cluster computing environment (see Figure 1.2) is a collection of homogeneous collocated desktops or servers interconnected by a Local Area Network (LAN) or by a high speed bus (*i.e.* blade clusters). Each cluster resource is called

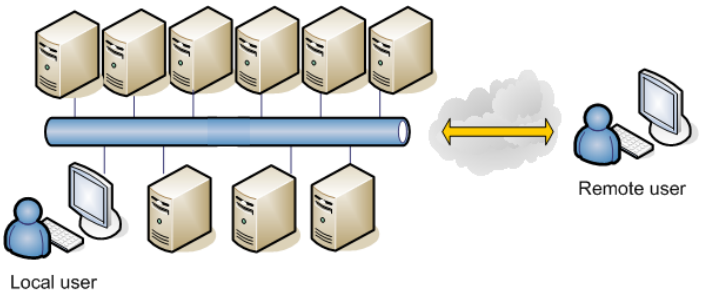


Figure 1.2: An example of a cluster architecture.

a "node" and it serves as a computational/storage resource for local or remote users running distributed applications. At first sight clusters have a lot in common with traditional supercomputers: supercomputers are also composed of multiple processors interconnected by a local high-speed computer bus; supercomputers as well as clusters are purchased and maintained by a single organization. However, clusters have a number of significant advantages that have made them probably the most popular High-Performance Computing (HPC) medium over the past 10 years. First of all, cluster nodes are relatively cheap and their acquisition can be spread in time to reduce costs. Secondly, it is much easier to upgrade a cluster capacity by adding extra nodes, to replace malicious nodes and to provide redundant components for security and availability reasons. The most well-known example of a cluster is the Beowulf-cluster [2] system that connects a number of Personal Computers (PC) running a Free and Open Source Software (FOSS) and a Unix-like Operating System (OS).

Grid computing (see Figure 1.3) is a distributed computing technology that was introduced middle 1990s [3]. It takes the principles of resource reuse and flexibility even further than cluster computing. Grids comprise heterogeneous world-wide distributed computational resources from different administrative domains. Grid resources can be dedicated machines, local clusters, supercomputers and desktops, all working together to achieve a common goal. The main principle of a computational grid is resource sharing: instead of purchasing redundant resources for a local cluster to cope with occasionally occurring peak loads, organizations can make use of idle resources all over the world, possibly subject to payment. This certainly cuts budgets for hardware infrastructure and increases resource utilization. To avoid that some organizations absorb most of the available grid resources, all grid users are subdivided into Virtual Organizations (VO). Resource sharing is allowed only between users belonging to the same VO. To guarantee smooth and reliable operation of grids, several important issues such as efficient resource utilization, fault-tolerance, security, Service Level Agreements (SLA), service ac-

Figure 1.3: An example of a grid architecture.

counting, *etc.* have to be addressed. Since most of the currently existing solutions for these issues are either too complex and time consuming or not sufficiently reliable, grids are until now mainly used in academic environments. One of the largest grid infrastructures in the world was built by the Enabling Grids for E-sciencE (EGEE) project [4]. The EGEE project led by the European Organization for Nuclear Research (CERN) currently incorporates 250 resource centers world-wide, providing about 40.000 CPUs and several Petabytes of storage. The infrastructure is mainly used to run High Energy Physics (HEP) applications.

Finally, cloud computing [5] (see Figure 1.4) is an emerging technology, which has in recent years received a growing interest from academic as well as commercial communities. Cloud computing can be compared to the electricity grid, which allows end-users to utilize the available resources, without requiring expertise or control of the underlying technology. While grid computing is mainly utilized for transparent processing of large computational tasks, cloud computing goes further by providing all kinds of dynamically scalable and virtualized resources (hardware, software and information) in the form of Internet services. Cloud services can be provided at three levels of abstraction: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). The first level typically delivers a platform virtualization environment as a service, which means that instead of purchasing servers, data space and network equipment a client can buy these resources as a fully outsourced service. The second level delivers a computing platform as a service, which often consumes cloud infrastructure and provides cloud applications. Finally, the third layer delivers software as a service over the Internet, eliminating the need to install, run and maintain applications on customer computers. IaaS and PaaS guarantee the most flexibility, but they also require the most maintanance and knowledge of underlying technologies. Cloud technology is confronted with the same issues as grid computing: fault-tolerance, security, load balancing, interoperability, scalable data storage, *etc*. Most of the currently existing cloud infrastructures are delivered through data centers, which build cloud services on top of their servers. Well-known cloud service providers are Ama-

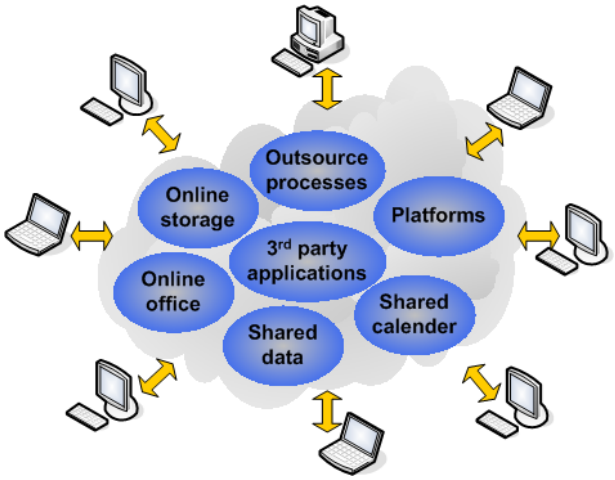


Figure 1.4: An example of a cloud architecture.

zon [6] and Google [7]. Amazon Elastic Compute Cloud (Amazon EC2) [8] is a PaaS web service that provides dynamically resizable computational capacity on the Amazon cloud. Users can scale up and down the number of server instances utilized, depending on changing application needs, while paying only for compute capacity they actually consume. The same principles are also implemented in Google App Engine [9], which is a PaaS enabling users to build and host web applications on the same systems as those powering Google applications.

1.1.2 Problems and Challenges

There are different challenges related to distributed execution of computationally intensive applications. These challenges are situated at the organizational level as well as at the level of software and hardware infrastructure. We summarize only the most prominent ones:

- **No clear standards**. There exists a broad variety of software for management of distributed resources. This software, usually referred to as "middleware", coordinates interaction between processes running within a distributed environment. Currently an insufficient number of common practices, agreements and specifications are defined to guarantee interoperability and reuse between different types of middleware.
- **Distributed system management**. This issue is mainly related to grid computing where heterogeneous devices are managed by different organizations. To construct a systematic framework for software, hardware and protocol administration, hierarchical management domains have to be defined.

- **Application gridification**. To gain advantage from distributed execution, applications have to be gridified or split into distributable components. While gridification of some applications is relatively straightforward, gridification of other applications is a complex and time consuming task. Unfortunately, it has to be performed manually, since no general automatic gridification procedure exists.
- **Software licensing**. Distributed execution requires application to be deployed to remote nodes, which leads to copyright and licensing issues.
- **Internal application dependencies**. Complex applications can contain interdependent tasks, which enforce a certain order on application execution. For many of the existing distributed systems, dependency preservation and efficient scheduling of tasks remain an open issue.
- **Centralized nature of middleware**. In many distributed computing systems we deal with centralized resource management infrastructures. These infrastructures usually consist of a number of servers performing one of the general functions, such as scheduling, monitoring, data storage *etc* that are crucial for the performance of distributed environments. A centralized infrastructure, however, often forms a performance bottleneck, limits the system scalability and creates a single point of failure.
- **Security**. This is a particularly prominent issue in a commercial context. Before sensitive data can be processed/stored on resources located outside an organization, users have to be sure that computational resources, data storage and network links are protected against malicious attacks.
- **Fault-tolerance**. Hardware failures of computational, storage or network resources within large distributed environments are commonplace. These should be handeled automatically without loss of important data and without noticable delays in application execution.

  In this work, several solutions for system fault-tolerance are proposed, which are based on the well-known replication and checkpointing techniques.
- **Scheduling**. A scheduling algorithm determines when and where applications are started within a distributed environment, while taking into account system status information, application properties and a scheduling objective. The following objectives are often employed:

  - ***CPU utilization***. A scheduler tries to increase CPU utilization and to reduce idle periods.
  - ***Throughput***. Maximization of the number of jobs that complete their execution within a predefined time unit.
  - ***Makespan***. Minimization of the time difference between the start and finish of a sequence of jobs or tasks.
  - ***Turnaround***. Minimization of the time interval between a job submission and its completion.

- ***Waiting time***. Minimization of waiting time of a job in a scheduler queue.
- ***Response time***. Minimization of the time interval between a job submission and availability of the first output.
- ***Fairness***. A scheduler ensures that each user gets equal processing time.
- ***Deadline preservation***. Guaranteeing the compliance of the schedule with predefined deadlines.
- ***Fault-tolerance***. Minimization of computational loss due to resource failures.

There exists a large variety of scheduling approaches for which different taxonomies are defined in literature [10–12]. In this thesis we differentiate between two categories: *static scheduling* and *dynamic scheduling*. By the term "static scheduling" we understand techniques with a predefined scheduling policy that does not change over time. Furthermore, once a job is scheduled by a static algorithm to a particular resource, it continues executing on the same resource until it completes or a resource failure interrupts its execution. On the other hand, *dynamic scheduling* comprises algorithms that can reassign jobs at run time, based on dynamically collected information on execution environment changes. Some dynamic algorithms are designed to adapt their internal scheduling policy in reaction to modifications of application execution patterns and changes in distributed environments (*e.g.* resource or link failure, arrival of new CRs, changing load on CRs). This group of algorithms is typically called *adaptive*. A dynamic or an adaptive scheduling approach is usually more sophisticated than a static mechanism. It requires extended monitoring facilities for distributed environments and applications as well as checkpointing and migration mechanisms. Obviously, execution of a complex scheduling algorithm, monitoring, periodic checkpointing and migration introduce a certain degree of computational overhead. This overhead should be compensated by the computational gain of intelligent scheduling, for an algorithm to be usable.

Design, optimization and evaluation of dynamic scheduling approaches form the main theme of this thesis.

1.2 Main Research Contributions

As was mentioned in the previous section, the design of efficient dynamic scheduling solutions is the main objective of this work. We consider two large topics particularly relevant for the operation of a large distributed environment running computationally intensive applications: fault-tolerance and scheduling of applications with dependencies. To cope with these issues we investigate several dynamic

solutions, whose efficiency is proven by simulation-based comparison with static approaches.

In a first instance, fault-tolerance of distributed environments is taken into consideration. We get a clear view on the effect resource failure has on the execution of applications with different duration by simulating and observing an unreliable grid environment. The grid is exposed to workloads with varying characteristics, for which parameters are derived from a real-world application. Furthermore, we consider two approaches commonly applied to deal with fault-tolerance: job replication and job checkpointing. When job replication is used, several copies ("replicas") of the same job are distributed to different CRs. If a failure or a computational slowdown occurs on some of the resources, the results from other resources can still be utilized to provide users with required outputs, ideally without introducing a noticeable delay. The more job replicas are executed, the lower the chance of computational loss or slow down. Checkpointing, on the other hand, periodically saves job status, in the form of so-called "checkpoints", to a failure-safe location, referred to as the Checkpointing Server (CS). To guarantee availability of checkpoints, this server is typically deployed in a redundant fashion. In case of a CR failure, affected jobs do not have to be re-run from the start but can instead be resumed from their latest checkpoint. Frequent checkpointing obviously requires less recomputations to be performed after a failure. A major problem with the replication and the checkpointing approaches is that they introduce significant computational overhead, if respectively an inappropriate number of replicas and exceedingly high checkpointing frequency are chosen. The optimal values for both parameters are hard to determine analytically, since they largely depend on resource load, size of checkpoints and the frequency of resource failure that can change dynamically over time. To automatically approximate these optimal values at any time during a job execution, we propose adaptive scheduling algorithms that modify the number of job replicas and the checkpointing frequency during the system operation, taking into account dynamic changes within the distributed environment.

In the next phase of this research, scheduling of applications composed of tasks with internal dependencies, also called "workflows", is studied. In particular, we consider input dependencies, whereby tasks can require inputs produced by other tasks before they can proceed with the execution. A dynamic scheduling algorithm for workflows is proposed that minimizes makespan of an application as a whole, at cost of potentially slower execution of individual tasks. The algorithm works for applications for which task execution progress can be monitored at run time. Using the progress information, predictions can be constructed on the total task processing time by means of extrapolation of the latest progress measurement. The execution time predictions calculated are applied to generate an efficient application schedule. Since resource load and internal application dynamics vary

over time, the algorithm periodically recalculates task execution time predictions and potentially reschedules already running tasks. However, rescheduling, or task migration, has to be performed cautiously since it can significantly delay task execution due to rescheduling overhead. Therefore, a number of possible optimizations of the algorithm are proposed, with as a goal the elimination of overzealous migrations. Finally, it is important to mention that the quality of execution time predictions largely determines the number of migrations performed. Since extrapolation-based predictions are very sensitive to internal application dynamics and variations in resource load, in the next phase, we propose a prediction method based on non-linear curve-fitting. The method matches historical data on extrapolated predictions against a number of predefined prediction evolution models. The "best-fit" coefficients obtained allow for more accurate prediction of task execution times.

To evaluate the performance of the algorithms discussed above, a novel grid simulation environment, called DSiDE (Dynamic Scheduling in Distributed Environments), was developed in the scope of this research. DSiDE has several advantages over other existing grid simulators: a simple and clear design; its speed of simulation execution; and its high flexibility with respect to modeling of dynamic system events, such as CR failure, CR load variations, changing task execution time predictions, *etc*. We choose grid systems for evaluation of the developed dynamic solutions, since resource failure and varying resource load are more prominently present in this type of computational environments. This can be explained by the internal complexity and highly distributed nature of grids. However, it is important to mention that the algorithms proposed also apply to other types of distributed systems. To provide realistic workload within our simulations, job parameters utilized were derived from an existing tool for modeling and virtual experimentation with environmental systems, called Tornado [13, 14]. Tornado forms an interesting use case for the evaluation of the developed dynamic solutions since it possesses a broad variety of jobs or "virtual experiments" with diverse characteristics in terms of computational complexity, dependencies, size of input and output, the possibility to monitor job progress, *etc*.

1.3 Important HPC systems

In this section we give a short overview of the most remarkable HPC systems. Some of them implement in one form or another fault-tolerance mechanisms and support for distributed execution of dependent tasks.

Globus Toolkit [15] is an open source software toolkit for building grids. It includes software services and libraries for resource discovery, management and monitoring, in addition to security services and data management. A noticeable flaw is the lack of Globus support for fault-tolerance. While an attempt was made

to foresee in a fault detection mechanism by means of the Heartbeat Monitor Interface (HBM) (is now disconnected as a separate module), the fault monitor alone is not sufficient to provide proper support for fault-tolerance, as long as tools and libraries essential for checkpointing and recovery are not available. Globus also does not provide support for scheduling of dependent tasks.

Condor [16] is a workload management system, mainly designed to coordinate job processing on clusters of homogeneous dedicated computational nodes. Condor provides support for intelligent cycle scavenging on idle desktop machines within a LAN. This interesting feature allows for more efficient resource utilization within an organization, without introducing any disadvantages for desktop users. When a user resumes his activity on a previously idle resource, user space is checkpointed and Condor jobs are canceled or migrated to a new idle resource. This approach can be seen as a form of resource-initiated dynamic scheduling. Considering fault-tolerance: Condor avoids computation loss through periodic kernel-level checkpointing and roll-back of affected jobs. Unfortunately, the Condor checkpointing library is only available for a limited number of software/hardware architectures, reducing the fault-tolerance support to a few platforms. Additionally, there exists a DAGMan (Directed Acyclic Graph Manager) [17] meta-scheduler for Condor that supports execution of jobs with simple as well as complex dependencies. However, there is no automatic job execution time prediction mechanism and all scheduling decisions are solely based upon best effort estimates provided by end users.

Condor and Globus technologies were combined in the **Condor-G** [18] project that allows multiple Condor computing environments to work together, forming a grid-like system. Condor-G combines advantages of both tools: it provides security and resource usage across domain boundaries, as supported within the Globus Toolkit; extensive job monitoring, logging, notification, fault-tolerance and scheduling of dependent tasks, are the features inherited from the Condor system.

LSF (Load Sharing Facility) [19] is a commercial workload management solution for HPC environments. It allows for scheduling of batch and interactive applications on clusters and grids. LSF contains user-level checkpointing libraries and supports kernel-level checkpointing on a limited number of platforms. When a failure occurs, users are provided with a choice to either restart their jobs from the latest checkpoints or from the beginning of their execution. LSF also implements several preemptive scheduling methods. Preemptive scheduling refers to job execution interruption to free resources for jobs with higher priority. After the high priority jobs terminate, the original jobs are restarted on the same resources. Finally, LSF supports a mechanism for specification of job dependencies and for preservation of their execution order.

PBS (Portable Batch System) [20] is a commercial resource management system for clusters, with an integrated scheduler. The software also has a limited open

source version, referred to as OpenPBS [21]. PBS does not provide user-level checkpoint libraries or support for job migration. However, kernel-level checkpointing is supported on several hardware platforms. Within PBS it is possible to specify job dependencies either by providing job execution order or by specifying particular execution conditions.

TORQUE (Terascale Open-Source Resource and QUEue Manager) [22] is an open source cluster management system that is based on the OpenPBS software. Compared to OpenPBS, Torque incorporates significant advances with respect to scalability, fault-tolerance and scheduling facilities. For instance, the middleware checks for additional failure conditions and these conditions are handled by restoring the affected jobs from their latest checkpoints. For improved utilization, scheduling and administration of clusters, Torque can be integrated with the Moab Workload Manager [23]. Moab is a commercial scheduler that optimizes cluster performance with intelligent resource allocation and workload ordering. It supports job prioritization, fairness policies, QoS, advanced reservation and job preemption. Moab does not provide direct support for job dependencies, but instead relies on the support by resource managers, such as Torque. Torque, in turn, allows basic dependencies specification through dynamically allocated linked lists, which are attached to jobs.

MOSIX [24] is a middleware solution for Linux clusters that can also operate on a multi-cluster scale. The middleware provides an advanced fully transparent dynamic load-balancing mechanism that monitors system state and attempts to improve the overall performance and provides fault-tolerance by dynamic resource (re)allocation. A unique feature of MOSIX is that it operates at the process-level, while most of the existing HPC systems operate at the job-level. This means that the system redistributes its workload when the number of processes changes, which is useful for load balancing of parallel jobs. On the other hand, process-level operation limits MOSIX in scheduling of dependent tasks, since there is no workflow notion.

LCG (LHC Computing Grid) [25] is a grid middleware developed by CERN, in the scope of the EGEE project [4], to process the immense amount of data generated by the Large Hadron Collider (LHC). LCG is based on the Globus Toolkit and adopts most of the services provided by Globus. LCG is a heavy-weight grid with huge unstable code base and relatively limited functionality. In particular, there are only limited job monitoring facilities, no support for any form of fault-tolerance, no possibility for dependencies presentation and preservation, *etc*. To address these issue a successor of LCG, called **gLite** [26], was introduced by EGEE. Fault-tolerance is provided in the new middleware by means of a resource failure monitoring facility in combination with automatic job resubmission. There are two kinds of resubmissions available in gLite: the *deep resubmission* and the *shallow resubmission*. The resubmission is *deep* when the job fails after it has

started running on a resource, and *shallow* otherwise. The maximum number of resubmissions can be defined on both types. gLite also supports job dependencies, which are represented using the Job Description Language (JDL), derived from the Classified Advertisement language (ClassAd) [27]. A flexible ClassAd language was initially introduced in the scope of the Condor project and it allows representation of dependencies of arbitrary structure.

OpenNebula [28] is an open source toolkit for cloud computing, providing mainly PaaS services. The toolkit maps user workload to the available physical resources, executed within Virtual Machines (VM), using one of several resource-aware allocation policies (*e.g.* packing, load-aware, affinity-aware, *etc*). Important to mention is that a recent part of the work performed in the scope of this thesis and discussed in [29], was dedicated to the efficient assignment of VMs within cloud environments. In concreto, we proposed an algorithm that assigns VMs according to the expected load of applications run within them. Apart from scheduling on the VM level, a workload assignment policy for multi-tier VMs, i.e. groups of interconnected VMs, is required. This functionality is not provided by OpenNebula but expected to be implemented at the application level. For this purpose, the algorithms proposed in the remainder of this work can be applicable. Finally, OpenNebula supports VM migrations and fault-tolerance, through a persistent database backend for storing VM information.

1.4 Thesis structure

To a large extent this thesis is organized based on a number of publications. The publications are selected to provide an integral and consistent overview of the work performed in scope of this PhD.

Chapter 2 introduces the DSiDE grid simulator: DSiDE is compared to a number of well-known simulators for grids, its architecture is clarified and input and output specifications are discussed. In Chapter 3, the effect of unexpected CR failure on application execution is studied. Chapter 4 proposes several adaptive algorithms to deal with the fault-tolerance issue. The algorithms are based on the task replication and the checkpointing concepts. Finally, scheduling of dependent tasks in presence of dynamic task progress information is addressed in Chapter 5, while Chapter 6 extends the algorithm developed in Chapter 5 with a curve-fitting-based task execution time prediction approach.

1.5 Publications

The PhD research performed resulted in several publications in scientific journals and in a series of presentations at international conferences. The following list

provides an overview of the publications.

1.5.1 A1: Publications indexed by the ISI Web of Science "Science Citation Index"

[1] **M. Chtepen**, F. Claeys, B. Dhoedt, P.A. Vanrolleghem, P. Demeester, *Computational complexity and distributed execution in water quality management*, published in Lecture Notes in Computer Science 3515, Proceedings of the 5th International Conference on Computational Science, MSN 2005, edited by V.S.Sunderam, G.D. van Albada, P.M.A. Sloot, J.Dongarra, ISBN: 978 – 3 – 540 – 26043 –1, Vol. LNCS 3515:1116–1119, Atlanta, GA, 22 – 25, May 2005

[2] F. Claeys, **M. Chtepen**, L. Benedetti, B. Dhoedt, P.A. Vanrolleghem, *Distributed virtual experiments in water quality management*, Water Science and Technology, ISSN 0273-1223, 53(1):297 – 305, Published by IWA Publishing, 2006

[3] **M. Chtepen**, F.H.A. Claeys, B. Dhoedt, F. De Turck, P. Demeester, P.A. Vanrolleghem, *Adaptive task checkpointing and replication: Towards efficient fault-tolerant grids*, IEEE Transactions on Parallel and Distributed Systems, ISSN 1045 – 9219, 20(2):180 – 190, Published by IEEE Computer Society, February 2009

[4] **M. Chtepen**, F.H.A. Claeys, B. Dhoedt, F. De Turck, P. Demeester, P.A. Vanrolleghem, *Dynamic Approach for Workflow Scheduling in Grids*, submitted to IEEE Transactions on Parallel and Distributed Systems

[5] **M. Chtepen**, F.H.A. Claeys, B. Dhoedt, F. De Turck, J. Fostier, P. Demeester, P.A. Vanrolleghem, *On-line Execution Time Prediction for Computationally Intensive Workflow Applications with Periodic Progress Updates*, submitted to the Journal of Supercomputing

1.5.2 P1: Publications indexed by the ISI Web of Science "Conference Proceedings Citation Index – Science"

[1] F. Claeys, **M. Chtepen**, L. Benedetti, B. Dhoedt, P.A. Vanrolleghem, *Distributed virtual experiments in water quality management*, The 6th International Symposium on Systems Analysis and Assessment in Water Management (WATERMATEX 2004), Beijing, China, November 3 – 4, 2004

[2] **M. Chtepen**, F.H.A. Claeys, B. Dhoedt, F. De Turck, P. Demeester, P.A. Vanrolleghem, *Dynamic scheduling of computationally intensive applica-*

tions on unreliable infrastructures, The 2nd European Modeling and Simulation Symposium (EMSS 2006), Barcelona, Spain, October 4 – 6, 2006

[3] F.H.A. Claeys, **M. Chtepen**, L. Benedetti, W. De Keyser, P. Fritzson, P.A. Vanrolleghem, *Towards transparent distributed execution in the Tornado framework*, The 2006 Environmental Application and Distributed Computing Conference (EADC 2006), Bratislava, Slovakia, October 16 – 17, 2006

[4] **M. Chtepen**, F.H.A. Claeys, B. Dhoedt, F. De Turck, P. Demeester, P.A. Vanrolleghem, *Evaluation of replication and rescheduling heuristics for grid systems with varying resource availability*, The 18th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2006), Dallas, TX, November 13 – 15, 2006

[5] **M. Chtepen**, F.H.A. Claeys, F. De Turck, B. Dhoedt, P.A. Vanrolleghem, P. Demeester, *Providing fault-tolerance in unreliable grid systems through adaptive checkpointing and replication*, The 7th International Conference on Computational Science (ICCS 2007), Beijing, China, May 27 – 30, 2007

[6] **M. Chtepen**, F.H.A. Claeys, F. De Turck, B. Dhoedt, P.A. Vanrolleghem, P. Demeester, *Scheduling of dependent grid jobs in absence of exact job length information*, The 4th IEEE/IFIP International Workshop on End-to-end Virtualization and Grid Management (EVGM 2008), Samos Island, Greece, September 22 – 26, 2008

[7] **M. Chtepen**, F.H.A. Claeys, B. Dhoedt, F. De Turck, P. Demeester, P.A. Vanrolleghem, *Adaptive checkpointing in dynamic grids for uncertain job durations*, The 31st International Conference on Information Technology Interfaces (ITI 2008), Dubrovnik, Croatia, June 22 – 25, 2009

[8] **M. Chtepen**, F.H.A. Claeys, F. De Turck, B. Dhoedt, P.A. Vanrolleghem, P. Demeester, *Performance evaluation and optimization of an adaptive scheduling approach for dependent grid jobs with unknown execution time*, The 18th International Congress on Modelling and Simulation (MODSIM 2009), Cairns, Australia, July 13 – 17, 2009

1.5.3 C1: Other Publications in International and National Conferences

[1] **M. Chtepen**, B. Dhoedt, P.A. Vanrolleghem, *Dynamic scheduling in grid systems*, published in 6th FTW PHD Symposium, Interactive poster session, paper nr. 110 (proceedings available on CD-Rom), Gent, Belgium, November 30, 2005.

[2] **M. Chtepen**, B. Dhoedt, P.A. Vanrolleghem, *Adaptive scheduling in grids*, published in 10th FTW PHD Symposium, Interactive poster session, pages 66 – 67 (proceedings available on CD-Rom), Gent, Belgium, December 9, 2009.

References

[1] University of California. *SETI@home Homepage*. http://setiathome.berkeley.edu/.

[2] Beowulf.org. *Beowulf Homepage*. http://www.beowulf.org/.

[3] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[4] Enabling Grids for E-science in Europe (EGEE). *EGEE: Enabling Grids for E-science in Europe*. http://egee-intranet.web.cern.ch/egee-intranet/gateway.html.

[5] B.P. Rimal, E. Choi, and I. Lumb. *A Taxonomy and Survey of Cloud Computing Systems*. In Proceedings of the 5th International Joint Conference on INC, IMS and IDC (NCM '09), Seoul, Korea, August 25 – 27 2009.

[6] Amazon.com company. *Amazon Web Services*. http://aws.amazon.com/.

[7] Google. *Google code Website*. http://code.google.com/intl/nl-BE/.

[8] Amazon.com company. *Amazon Elastic Compute Cloud Website*. http://aws.amazon.com/ec2/.

[9] Google. *Google App Engine Website*. http://code.google.com/intl/nl-BE/appengine/.

[10] T.L. Casavant and J.G. Kuhl. *A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems*. IEEE Transactions on Software Engineering, 14(2):141 – 154, February 1988.

[11] A. Andrieux, D. Berry, J. Garibaldi, S. Jarvis, J. MacLaren, D. Ouelhadj, and D. Snelling. *Open Issues in Grid Scheduling*. Technical report, National e-Science Centre, October 2003.

[12] Y. Liu. *Grid Scheduling*. Technical report, Department of Computer Science, University of Iowa, 2004.

[13] F. Claeys, D.J.W. De Pauw, L. Benedetti, I. Nopens, and P.A. Vanrolleghem. *Tornado: a Versatile and Efficient Modelling & Virtual Experimentation Kernel for Water Quality Systems Applications*. In Proceedings of the International Environmental Modelling and Software Conference (iEMSs), Burlington, Vermont, USA, July 9–13 2006.

[14] F.H.A. Claeys. *A Generic Software Framework for Modelling and Virtual Experimentation with Complex Environmental Systems*. Phd thesis, Ghent University, Department of Applied Mathematics, Biometrics and Process Control, Coupure Links 653, B-9000 Gent, Belgium, January 2008.

[15] The Globus Alliance. *The Globus Website*. http://www.globus.org/.

[16] The University of Wisconsin. *The Condor Project Website*. http://www.cs.wisc.edu/condor/.

[17] Condor Project. *DAGMan (Directed Acyclic Graph Manager) Website*. http://www.cs.wisc.edu/condor/dagman/.

[18] The University of Wisconsin. *The Condor-G Project Website*. http://www.cs.wisc.edu/condor/condorg/.

[19] Platform Computing Corporation. *Load Sharing Facility (LSF) Website*. http://www.platform.com/workload-management/high-performance-computing/.

[20] Inc. Altair Engineering. *PBSWprks Enabling On-Demand Computing*. http://www.pbsworks.com/.

[21] Math & Computer Science Division of Argonne National Laboratory. *OpenPBS Public Home*. http://www.mcs.anl.gov/research/projects/openpbs/.

[22] Inc. Cluster Resources. *TORQUE Resource Manager Website*. http://www.clusterresources.com/products/torque-resource-manager.php.

[23] Inc. Cluster Resources. *Moab Workload Manager Website*. http://www.clusterresources.com/products/moab-cluster-suite/workload-manager.php.

[24] A. Barak and A. Shiloh. *MOSIX: Cluster and Multi – Cluster Management*. http://www.mosix.org/.

[25] LHC Computing Project. *LCG (LHC Computing Grid) Website*. http://lcg.web.cern.ch/LCG/.

[26] EGEE (Enabling Grids for E-sciencE). *gLite Lightweight Middleware for Grid Computing Website*. http://glite.web.cern.ch/glite/.

[27] Condor Project. *Classified Advertisements*. http://www.cs.wisc.edu/condor/classad/.

[28] OpenNebula Project. *OpenNebula.org*. http://www.opennebula.org/.

[29] L. Deboosere, M. Chtepen, and B. Dhoedt. *A Computational Load Limit-Based Algorithm for Thin Clients*. In preparation.

2
DSiDE Simulator

This chapter introduces the DSiDE grid simulator, which was developed in the context of this PhD research. DSiDE was used to evaluate the performance of the designed dynamic scheduling solutions and to compare these to a number of existing static approaches. In this chapter the following topics are successively discussed: comparison of existing simulators with DSiDE, DSiDE design and architecture, input and output specification.

2.1 Existing HPC Simulators

It is not always financially and technically feasible to build a realistically-sized grid testbed for evaluating the performance of dynamic scheduling algorithms. First of all, the majority of the existing grids are operational and cannot be used for experimental purposes, while building a separate research grid requires significant investments for hardware infrastructure and support staff. Secondly, in a real testbed it is difficult to achieve a controlled and repeatable sequence of events, often required to test algorithms under certain conditions. Therefore, grid simulators provide a cheap, fast and relatively accurate means for algorithm performance measurements.

We evaluated a number of well-known discrete-event simulators with respect to their extensibility, scalability, efficiency, ability to model grid components and dynamic grid behavior. These simulators can be subdivided into two groups: general purpose simulators (*e.g.* GPSS [1–3], SLX [1, 4]) and grid specific simulators (*e.g.*

GridSim [5, 6], SimGrid [7], NSGrid [8], HyperSim-G [9] and ChicSim [10]). Table 2.1 shows a qualitative comparison between different simulation environments, including DSiDE.

From the evaluation study performed, it can be concluded that general purpose simulators are usually easy to learn and to use due to the extended documentation and time-aware debugging mechanisms available. Furthermore, they are fast and easily extensible. Their main disadvantages are the difficulty of implementation of dynamic components and the lack of built-in models for relevant grid and network resources.

In contrast to general purpose simulators, grid-specific simulators possess built-in grid models. However, the models are often not flexible enough to be extended with additional functionality. Furthermore, most of the simulators do not support modeling of dynamic resource and job behavior, have long simulation times and limited scalability. Also, the absence of detailed documentation and debugging facilities complicates their usage.

To address the issues of existing simulators discussed above, DSiDE was conceived. DSiDE is a discrete-event simulator that provides built-in models for the most relevant grid components. Thanks to its general and clear architecture, the simulator can easily be extended with new, not necessarily grid-related, models. The main advantage of DSiDE is its extended support for easy and flexible modeling of all kinds of dynamic grid behavior. The simulator is also relatively fast, partly thanks to the selection of appropriate implementation technologies and partly thanks to design choices made with respect to network implementation. The latter requires only a limited number of events to model sufficiently accurate network delays. So far, the scalability of DSiDE was tested with up to 1,000 nodes, running several thousands of jobs simultaneously. However, the scalability is determined not only by the size of the grid simulated, but also by the dynamics of the simulation. In particular, the number of events processed is the main factor influencing the speed and the scalability of DSiDE. Finally, there is a possibility to debug DSiDE through an Integrated Development Environment (IDE), but there is no DSiDE specific debugging facility. Also the currently available documentation is rather limited.

2.2 Design of DSiDE

In general, the simulator built can be described as a flexible and portable framework for modeling and simulation of distributed computing environments. The term “flexible” refers to the fact that although currently DSiDE primarily contains built-in models for grid components, the simulator can be easily extended for a wide variety of problems. DSiDE is also “portable” since it is available for win32, win64, linux and potentially other platforms. The simulator was developed in

	GPSS	SLX	GridSim	SimGrid	NSGrid	HyperSim-G	ChicSim	DSiDE
Built-in grid models	-	-	+	+	+	+	+	+
Dynamic behavior modeling	-	+	-	-	+	-	-	+
Built-in network models	-	-	+/-	+/-	+	+/-	-	+/-
Extensibility	+	+	+	+	+/-	+	-	+
Genericity	+	+	+	-	-	+	-	+
Scalability	-	+/-	-	-	-	+	+	+
Simulation speed	+	+	-	+	-	+	-	+
External code integration	-	+	+	+	+	+	-	+
Documentation	+	+	+/-	+	-	-	-	-
Portability	+	-	+	+	-	-	+	+
Time-aware debugging	+	+	-	-	-	-	-	-
Modeling language	GPSS	SLX	Java	C	Tcl	C++	Prasec	XML
Freely available	+	+/-	+	+	+	+	-	+

Table 2.1: Discrete-event simulator comparison: “+” indicates that a feature is supported; “+/-” indicates that a feature is partially supported; “-” indicates that a feature is not supported.

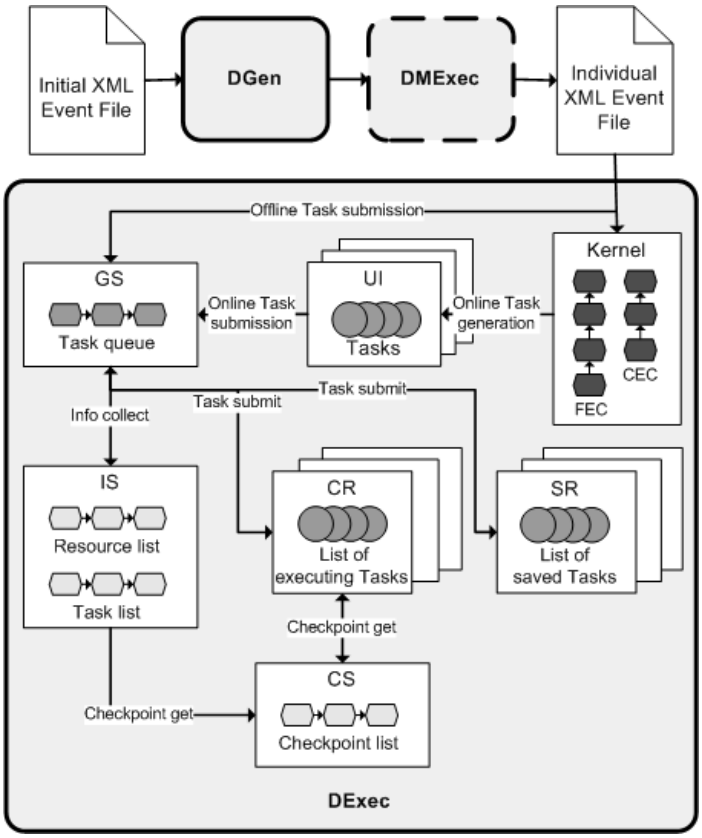


Figure 2.1: DSiDE conceptual diagram.

C++, as this object-oriented programming language offers a good compromise between efficiency and advanced features such as high-level data constructs, abstract interfaces, smart pointers, exception handling, namespaces, *etc*.

The architecture of DSiDE consists of three separate modules: DGen, DMExec, and DExec (see Figure 2.1). Simulation scenarios and grid models are specified in DSiDE through an initial XML event file that contains descriptions of recurrent event types and the way in which they should be generated (offline or at run-time). DGen translates this initial specification into individual events that can be loaded into the DExec module. At the heart of DExec is the Kernel component, which maintains two event chains: the Future Event Chain (FEC) and the Current Event Chain (CEC). All simulation initialization events and events representing dynamic grid changes arrive into the FEC where they are sorted by their simulated timestamp. During each iteration, the first event in the FEC is moved to the CEC to-

gether with all other events with the same timestamp. Afterwards, the events in the CEC can be sorted and executed in any predefined order. The simulator keeps running until either the end event is executed, or the simulation end-time is reached. Next to the Kernel, DExec implements a grid simulation environment, containing models for the following grid elements: Computational Resource (CR), Storage Resources (SR), User Interface (UI), Grid Scheduler (GS), Information Service (IS) and Checkpointing Server (CS). CRs and SRs are the actual grid resources, where respectively jobs are executed and inputs/outputs are stored. The capacity of CRs is measured in Million Instructions Per Second (MIPS) and of SRs in KBytes. Two CR models are implemented in DSiDE: in the first model the capacity of each CR is equally divided between all jobs running on the resource; in the second model a percentage of the total CR capacity (determined by job requirements) is assigned to each arriving job, until the resource get filled. While the second model is rather straightforward, Figure 2.2 gives an example for the first model. When a job is scheduled to a resource by a GS, the resource remains reserved for the job until the input data transfer terminates. Afterwards, the job starts running, sharing the resource capacity with other active jobs. In the example a CR initially executing a single job J_1 is considered. J_1 running on the resource with speed MIPS=1 will execute during 6 time units ($E_{J_1} = 6$). When the second job J_2 arrives, the resource speed is divided by 2 (MIPS=0.5) and the remaining execution time of J_1 is prolonged by the factor of 2. Similarly, when the third job J_3 arrives, the remaining execution times of the first two jobs are prolonged with the factor of 3, as the total computational capacity is now divided among 3 jobs. When J_1 execution ends, the remaining execution times of J_2 and J_3 are reduced to reflect the fact that MIPS per job increase from 0.33 to 0.5. When job outputs are transferred to some SR, it is assumed that the job is not longer consuming computational resources. Therefore, the computational speed of concurrently running jobs increases and another job can eventually be started on the resource. The model for SR, in turn, assumes a certain storage capacity that is decreased each time new data has to be stored.

The UI is a grid service responsible for the dynamic generation of jobs with particular characteristics, which are submitted to the GS with a specified frequency. Grid models can contain an unlimited number of UIs. In contrast to UIs, DSiDE currently supports only a centralized scheduling model, where a single GS is responsible for distribution of all arriving jobs to the available resources, according to a particular scheduling strategy. Currently, various scheduling algorithms are implemented in DSiDE. They can be subdivided into the following categories:

- **Benchmark**: algorithms discussed in literature that have been used as benchmarks.

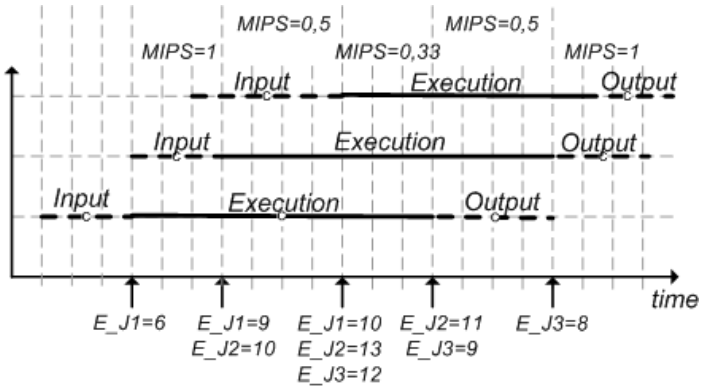


Figure 2.2: CR capacity sharing model.

- ***First Come First Served (FCFS)***: schedules the longest waiting jobs in the GS job queue to a random resource. Only available resources are considered for job execution, where the term "available resource" refers to a non-failed CR, with less than n_{CR}^{max} jobs running. On an available resource each job is assigned an equal amount of hardware resources (CPU, IO bus, *etc.*).
- ***First Job Best Computational Resource (FJBCR)***: assigns jobs to CRs in the order of their arrival into the GS job queue. Each job is assigned to the fastest available resource CR, *i.e.* a resource with the highest speed ($MIPS_{CR}$) and the smallest number of jobs (n_{CR}) currently scheduled to this CR.
- ***MaxMax***: assigns the longest (*i.e.* with longest execution time) idle job in the GS job queue to the fastest available resource.
- ***MinMax***: assigns the shortest idle job in the GS job queue to the fastest available resource.
- ***Unconditional Replication (UnconditionalRL)***: assigns a predefined number (Rep) of job replicas to preferably widely distributed CRs, to reduce the chance of simultaneous failure. When all resources are occupied, job replication is postponed.
- ***Workqueue with Replication (WQR)***: distributes the first job copy to a random idle resource in FCFS order. When the job queue is empty and the system has free resources, replication is activated.
- ***Failure Detection and Migration***: in case of a resource failure detection, restarts affected jobs from the their last checkpoints.
- ***Rescheduling***: considers the situation when CR load dynamically varies over time. Each job is initially scheduled to the fastest available resource in FCFS order. In each scheduling iteration, system status is evaluated and jobs are eventually rescheduled to faster available resources.

- **Fault-tolerant scheduling**: algorithms developed to provide system fault-tolerance (see Chapter 4).

  - ***Load Dependent Replication (LoadDependentRep)***: performs replication only when there are a sufficient number of idle CRs available within a distributed environment. When the system is overloaded and a predefined idle CPU limit is reached, replication is postponed until the system load decreases.
  - ***Failure Detection and Replication***: combines job failure detection and restart with unconditional replication.
  - ***Failure Detection and Load Dependent Replication (FailureDependentRep)***: combines job failure detection and restart with load dependent replication.
  - ***Replication and Migration (CombinedFT)***: alternates between job replication and checkpointing. As long as enough CRs are available, job replication is performed. When the system gets overloaded, the algorithm switches to job checkpointing.

- **Workflow scheduling**: algorithms that address scheduling of jobs composed of dependent tasks (see Chapter 5).

  - ***Workflow Scheduling***: schedules jobs composed of internal tasks organized into a workflow. Jobs are processed in FCFS order, while tasks are executed in the order indicated by their dependency graph. Each task is assigned to the fastest available resource.
  - ***Balanced Workflow Scheduling***: balances execution times of tasks having the same *dependents*, so that the execution time of the longest task is minimized and the other tasks finish more or less simultaneously with the longest task. A *dependent task* is a task that depends on data generated by other tasks to proceed with its execution. The idea of the algorithm is to optimize resource utilization (by scheduling tasks to resources with minimum required speed) and at the same time to decrease execution time of workflows. The algorithm is able to deal with highly dynamic computational environments, where resource speed and expected task execution time vary over time. Checkpointing and task rescheduling are applied to address these dynamic variations.
  - ***Balanced Workflow Scheduling with Resource Grouping***: the algorithm operates similar to the previous one, except that instead of matching each task to each available resource, tasks are matched against groups of CRs to improve scalability of the algorithm. CRs are grouped based on their location and speed, and the average parameters of each group are utilized by the matchmaking algorithm.

  - ***Balanced Workflow Scheduling with Reduced Overhead***: the algorithm operates similar to "Balanced Workflow Scheduling" but to reduce the migration overhead, workflows are rescheduled only when expected remaining tasks execution times are sufficiently long and when significant profit is expected to be achieved by migration.

- **VM scheduling**: algorithms that address scheduling of interactive session-based applications within clouds (see [11]).

  - ***Capacity Matching and User Clustering***: deals with user-specific applications with varying, *a priori* known requirements with respect to resource computational capacity. The algorithm assigns jobs based on their origin, which means that jobs from the same user are scheduled together, as much as possible. Therefore, resources are processed in order of the decreasing number of running jobs originating from the same user and the next job is assigned to the first processed resource with sufficient capacity. This algorithm does no support a waiting queue but instead jobs are either scheduled immediately after their arrival or rejected and lost forever.
  - ***Capacity Matching Full Fill***: similar to the previous algorithm, but jobs are no longer clustered according to their user. Instead, CRs are filled as much as possible with consecutively arriving jobs in order of their appearance in the IS resource queue.
  - ***Capacity Matching Full Fill Round Robin***: analogous to the previous approach, but resources are considered for job execution in Round Robin (RR) order.
  - ***Capacity Matching with Limit***: operates on the same type of applications as "Capacity Matching and User Clustering". The algorithm iterates over resources in RR order and only schedules jobs that have capacity requirements below a predefined resource-bounded limit. All jobs with resource requirements above the limit are rejected.
  - ***Capacity Matching with Dynamic Limit***: similar to the previous algorithm, except that the resource-bounded limit dynamically varies over time. If a sufficient number of "light" (*i.e.* with capacity requirements below the limit) jobs are running on a resource, its limit can be dynamically increased to allow processing of "heavy" jobs (*i.e.* with capacity requirements above the limit) and *visa versa*. The Erlang B formula [12] is used to calculate the limit values at run time, based on job arrival frequency and processing times.

Each of the above mentioned scheduling approaches can be provided with various input parameters that can be used for further refinement of a scheduling policy.

Scheduling decisions taken by the algorithms are always based on information collected from the IS in each scheduling iteration. At this moment only one IS can

be defined, which can function in two modes: changes in system status are propagated to the IS immediately; the IS requests status from resources periodically using a constant time interval I_{IS}.

As was mentioned earlier, DSiDE also supports checkpointing of jobs. Three checkpointing strategies are currently implemented in DSiDE:

- **Periodic Checkpointing (PeriodicCP)**: jobs/tasks are checkpointed using a predefined constant time interval (I).
- **Last Failure Dependent Checkpointing (LastFailureCP)**: periodic checkpoints are saved / omitted depending on the estimated chance of a job (or a task) to fail during the next interval I. This chance is estimated based on total job execution time and on time expired since the last failure of the CR, where the job is assigned.
- **Mean Failure Dependent Checkpointing (MeanFailureCP)**: the checkpointing interval I is enlarged / reduced depending on the total job execution time and on the measured mean failure frequency of the CR, where the job is assigned.

Multiple CSs can be modeled for checkpoint storage. However, since data management is out of scope of this dissertation, no strategies are implemented for intelligent allocation of checkpointing data between different CSs.

It is important to mention that DSiDE provides a set of events to specify network links and routes (sequence of links), which form the network model of the simulator. The DSiDE network is similar to the SimGrid network model, which differentiates between two types of links: WAN or Internet links; and LAN or intra-site links. WAN links are assumed to be fully interconnected with equal bandwidth assigned to each route going through them (a small fraction of the total bandwidth). On the other hand, intra-site links are always organized into a tree topology and the available bandwidth is proportionally shared among the simultaneous active data transfers [13]. This simple model has proven to be a good approximation for real network behavior [14], while preserving relatively low computational complexity and short simulation times.

Finally, the DMExec module of DSiDE is used to either execute a batch of predefined simulation experiments automatically or to run the simulation with different seed values for the Random Number Generator (RNG). A RNG is used within DSiDE to generate event occurrences and parameter values according to selected distributions. Successively initializing RNG with different seeds, results in different sequences of pseudo random numbers and provides us with a clear view on the variance of the obtained simulation results. If the variance is relatively high, it suggests that the system experiences difficulties to reach a single steady state situation, for example due to an insufficient number of job runs or due to an insufficiently long simulation period considered, or due to inherent bi-stability.

2.3 DSiDE Input

Models and simulation scenario descriptions are specified in DSiDE by means of an XML-input file of the form shown below. Input to DSiDE is composed of a sequence of events, located under the tag Events. Each individual event within the Events-tag contains a description of recurrent events to be simulated. The arguments "Type" and "Number" indicate respectively the type of events and the number of event instances. For instance, instantiation of CRs with similar capacity, similar activation times, located within the same LAN and having the same load variation pattern can be specified as one event, indicating the "arrival" of the required number of similar CR instances into a grid. Properties within Event-tags stand for static component characteristics (*e.g* name, location, user name), as well as dynamic component behavior (*e.g* CR load variation, job execution progress variation, variation in job submission pattern), described by different RNG distributions (Uniform, Normal, LogNormal, Poisson, Weibull, *etc.*).

```
<DSiDE>
 <Events Version="1.0">
  <Event Type="Sched.Add">
   <Description>
    <Props>
     <Prop Name="Number" Value="1"/>
     <Prop Name="RegisterTime" Value="0"/>
      ...
    </Props>
    <Algorithm Name="CostAwareRescheduling">
     <Props>
      <Prop Name="Checkpoint" Value="true"/>
       ...
     </Props>
    </Algorithm>
   </Description>
  </Event>

  <Event Type="Job.Add" Number="0">
   <Description>
    <Props>
     <Prop Name="SubmissionType" Value="Dynamic"/>
     <Prop Name="SubmissionFrequency" Value="RNG0"/>
      ...
    </Props>
    <Distributions>
     <RNG>
      <Props>
       <Prop Name="Name" Value="RNG0"/>
       <Prop Name="Distribution" Value="Uniform"/>
       <Prop Name="LBound" Value="100"/>
```

```
      <Prop Name="UBound" Value="100"/>
     </Props>
    </RNG>
    ...
   </Distributions>
  </Description>
 </Event>

 <Event Type="Resource.Add" Number="4">
  <Description>
   <Props>
    <Prop Name="Name" Value="CR"/>
    <Prop Name="RegisterTime" Value="0"/>
    ...
   </Props>
       ...
  </Description>
 </Event>

 ...

</Events>
</DSiDE>
```

This short form specification is converted into a list of individual events, suitable for execution by DExec, using the DGen Command-Line-Interface (CLI):

```
dgen [options] <XMLEventsFile>

Options:
-h,  --help          Show this message.
-o,  --out <File>    Specify output file name.
```

This interface requires as input the name of a file ("-o" option) where the converted list of events can be stored and the name of the input XML-file. An example of a list of converted Events is shown below:

```
<DSiDE>
<Events Version="1.0">
<Event Type="Network.Add" Time="0">...</Event>
<Event Type="Sched.Add" Time="0">...</Event>
<Event Type="Sched.Run" Time="0">...</Event>
<Event Type="Job.Add.Dynamic" Time="1">...</Event>
<Event Type="CR1.Add" Time="0">...</Event>
<Event Type="CR2.Add" Time="0">...</Event>
<Event Type="CR3.Add" Time="0">...</Event>
<Event Type="CR4.Add" Time="0">...</Event>
...
</Events>
</DSiDE>
```

The output of DGen can either be used to run a single simulation within the DExec module or multiple simulations, using the DMExec CLI. DMExec is called using the following command:

```
dmexec [options] <ExecutionMode> <StopMode> <XMLEventsFile>

Options:
-h,  --help          Show this message.
-l,  --log <File>    Specify log file name.
-o,  --out <File>    Specify output file name.
-ml, --mlog <File>   Specify common log file name.
```

DMExec can be run in two different execution modes:

- **Batch**: several numbered input files can be automatically processed by DExec one after another. This mode is run to speed up execution of a batch of simulations on a local host, where only one simulation can be run simultaneously at reasonable speed. The "-en" argument (is not an option and is not specified in the option list above) specifies the number of simulations in a batch (N_{batch}). The names of an input files for each simulation should have a prefix "XMLEventsFile" followed by a sequential number from 0 to N_{batch}, to be recognized and executed by DMExec.
- **Seed**: runs the same simulation experiments with varying seeds. The "-sn" argument specifies the number of different seeds (N_{seed}) the simulation should be run with. By default, a seed is incremented with 1 in each iteration. Low variance between the results obtained with different seeds, indicates that the steady state is reached within a simulation experiment.

There are also two modes to specify the end condition for simulations incorporated into DMExec:

- **NoJobs**: simulation ends when the number of jobs, indicated by the "-j" argument, is successfully executed.
- **Time**: simulation ends when the simulated time, indicated by the "-t" argument, is reached.

Next to logs and output files per simulation (respectively "-l" and "-o" options), DMExec produces an output file ("-ml" options) with statistics on the results of all simulation experiments run.

The CLI of DExec is more or less similar to the previous one, except that it does not include options for multiple simulation runs:

```
dexec [options] <StopMode> <XMLEventsFile>

Options:
-h,  --help          Show this message.
-l,  --log <File>    Specify log file name.
-o,  --out <File>    Specify output file name.
```

2.4 DSiDE Output

DSiDE generates three types of output: output files with statistics on the results of batch simulations; log files containing information on Events executed during one simulation; and output files including statistics on jobs and resources during one simulation run.

A common output file is a text file of the following form:

```
I Jan 07 14:35:09 2010   Run 0: JobDone=4554
I Jan 07 14:35:09 2010   Run 0: AccJobStableState=4602
I Jan 07 14:35:09 2010   Run 0: RejJobStableState=1057
I Jan 07 14:35:09 2010   Run 0: CPULoadAccJobs=9.144
I Jan 07 14:35:09 2010   Run 0: SubmitJobsUnderLimit=3027
I Jan 07 14:35:09 2010   Run 0: AccJobsUnderLimit=2792
I Jan 07 14:35:09 2010   Run 0: AccJobsUnderLimitPr=0.922
I Jan 07 14:35:09 2010   Run 0: SubmitJobsAboveLimit=2632
I Jan 07 14:35:09 2010   Run 0: AccJobsAboveLimit=1810
I Jan 07 14:35:09 2010   Run 0: AccJobsAboveLimitPr=0.687
I Jan 07 14:35:09 2010   Run 0: SystemLoad=0.645
I Jan 07 14:35:09 2010   Run 0: JobsOnCRAvg=9.101
I Jan 07 14:35:09 2010   Run 0: CPULoadAvg=76.681
I Jan 07 14:35:09 2010   Run 0: LongJobsProfit=0
I Jan 07 14:35:09 2010   Run 0: R-limit=0
I Jan 07 14:35:34 2010   Run 1: JobDone=3492
I Jan 07 14:35:34 2010   Run 1: AccJobStableState=3531
I Jan 07 14:35:34 2010   Run 1: RejJobStableState=727
I Jan 07 14:35:34 2010   Run 1: CPULoadAccJobs=9.663
I Jan 07 14:35:34 2010   Run 1: SubmitJobsUnderLimit=2250
I Jan 07 14:35:34 2010   Run 1: AccJobsUnderLimit=2056
I Jan 07 14:35:34 2010   Run 1: AccJobsUnderLimitPr=0.913
I Jan 07 14:35:34 2010   Run 1: SubmitJobsAboveLimit=2008
I Jan 07 14:35:34 2010   Run 1: AccJobsAboveLimit=1475
I Jan 07 14:35:34 2010   Run 1: AccJobsAboveLimitPr=0.734
I Jan 07 14:35:34 2010   Run 1: SystemLoad=0.524
I Jan 07 14:35:34 2010   Run 1: JobsOnCRAvg=7.884
I Jan 07 14:35:34 2010   Run 1: CPULoadAvg=64.298
I Jan 07 14:35:34 2010   Run 1: LongJobsProfit=0
I Jan 07 14:35:34 2010   Mean=4066.5 Variance=573520.5
I Jan 07 14:36:04 2010   Run 2: JobDone=4035
I Jan 07 14:36:04 2010   Run 2: AccJobStableState=4054
I Jan 07 14:36:04 2010   Run 2: RejJobStableState=783
I Jan 07 14:36:04 2010   Run 2: CPULoadAccJobs=9.373
I Jan 07 14:36:04 2010   Run 2: SubmitJobsUnderLimit=2591
I Jan 07 14:36:04 2010   Run 2: AccJobsUnderLimit=2433
I Jan 07 14:36:04 2010   Run 2: AccJobsUnderLimitPr=0.939
I Jan 07 14:36:04 2010   Run 2: SubmitJobsAboveLimit=2246
I Jan 07 14:36:04 2010   Run 2: AccJobsAboveLimit=1621
I Jan 07 14:36:04 2010   Run 2: AccJobsAboveLimitPr=0.721
```

```
I Jan 07 14:36:04 2010   Run 2: SystemLoad=0.572
I Jan 07 14:36:04 2010   Run 2: JobsOnCRAvg=8.104
I Jan 07 14:36:04 2010   Run 2: CPULoadAvg=71.819
I Jan 07 14:36:04 2010   Run 2: LongJobsProfit=0
I Jan 07 14:36:04 2010   Mean=4062.3 Variance=286812.3
I Jan 07 14:36:04 2010   ******************************
I Jan 07 14:36:04 2010   JobDoneMean=4027
I Jan 07 14:36:04 2010   AccJobMean(stable st.)=4062.333
I Jan 07 14:36:04 2010   RejJobMean(stable st.)=855.666
I Jan 07 14:36:04 2010   AccJobMeanPr=0.826
I Jan 07 14:36:04 2010   AccJobsUnderLimitMeanPr=0.925
I Jan 07 14:36:04 2010   AccJobsAboveLimitMeanPr=0.714
I Jan 07 14:36:04 2010   SystemLoadPr=0.580
I Jan 07 14:36:04 2010   JobsOnCRAvg(stable st.)=8.363
I Jan 07 14:36:04 2010   CPULoadAvg(stable st.)=70.933
I Jan 07 14:36:04 2010   TotalLongJobsProfit=0
```

The file is generated during the execution of DMExec and has a free format. It contains the desired statistics per simulation run, as well as general statistics for all simulations within a batch. New statistics can be added or old statistics can be removed, but this requires DMExec to be recompiled. Furthermore, each line of the file specifies either it contains a standard output ("I" symbol) or an error ("E" symbol) message and the timestamp of the generation of each message is provided.

A simulation log file of the DExec module is also a free style output file that is utilized for debugging purposes as it contains information on all Events executed by DExec during each simulation:

```
I Apr 16 20:50:28 2010   Loading: Test.Exec.Events0.xml
I Apr 16 20:50:29 2010   0: (Event=2): Network.Add:...
I Apr 16 20:50:29 2010   0: (Event=3): Routing.Add:...
I Apr 16 20:50:29 2010   0: (Network): Route.Add:...
I Apr 16 20:50:29 2010   0: (Network): Route.Add:...
I Apr 16 20:50:29 2010   0: (Network): Route.Add:...
I Apr 16 20:50:29 2010   0: (Network): Route.Add:...
I Apr 16 20:50:29 2010   0: (Event=4): Sched.Add:...
I Apr 16 20:50:29 2010   0: (Event=5): Sched.Run: Sched0
I Apr 16 20:50:29 2010   0: (Event=6): IS.Add:...
I Apr 16 20:50:29 2010   0: (Network): Route.Add:...
I Apr 16 20:50:29 2010   0: (Event=7): CS.Add:...
I Apr 16 20:50:29 2010   0: (Event=9): CR.Add:...
I Apr 16 20:50:29 2010   0: (Network): Route.Add:...
I Apr 16 20:50:29 2010   0: (Network): Route.Add:...
I Apr 16 20:50:29 2010   0: (IS0): ResourceAdd CR0

...
```

Finally, after each simulation run DExec generates an XML-output file (see listing below). Similar to the input file, the Events-tag specifies output events.

Individual output events contain extensive statistical information, grouped per resource type: CR (availability, average load,*etc.*), SR (average disk space occupied, *etc.*), network (amount of input and checkpointing data transferred, *etc.*), distributed system in general (number of jobs processed, characteristics of executed jobs, *etc.*).

```
<DSiDE>
 <Events Version="1.0">
  <Event Type="Jobs.Output"/>
  <Event Type="ResourceStatistics.Output">
   <Description>
    <Resource>
     <Props>
      <Prop Name="Name" Value="CR0"/>
      <Prop Name="FailureTime" Value="0"/>
      <Prop Name="Availability" Value="100"/>
      <Prop Name="ProcessedInstr" Value="146629211"/>
      <Prop Name="NonIdleTime" Value="44635515"/>
      <Prop Name="NonIdleTimePr" Value="51.661"/>
      <Prop Name="FullyLoadedTime" Value="44635515"/>
      <Prop Name="FullyLoadedTimePr" Value="51.661"/>
     </Props>
    </Resource>

   ...

   </Description>
  </Event>
  <Event Type="NetworkStatistics.Output">
   <Description>
    <JobAllocationPath>
     <Props>
      <Prop Name="Name" Value="N3:N1"/>
      <Prop Name="JobAllocated" Value="6991"/>
      <Prop Name="JobAllocatedPr" Value="100"/>
     </Props>
    </JobAllocationPath>
    <NetworkPath>
     <Props>
      <Prop Name="Name" Value="N3:N1"/>
      <Prop Name="NetworkDataTransfer" Value="209730"/>
     </Props>
    </NetworkPath>
   </Description>
  </Event>
  <Event Type="SystemStatistics.Output">
   <Description>
    <Props>
```

```
      <Prop Name="JobTotal" Value="9724"/>
      <Prop Name="JobDoneNoDouble" Value="6986"/>
      <Prop Name="JobDoneDouble" Value="6986"/>
      <Prop Name="FinalJobDone" Value="6986"/>
      <Prop Name="JobLost" Value="0"/>
      <Prop Name="JobSysTimeMean" Value="4249030132.202"/>
      <Prop Name="JobExecTimeMean" Value="37551.124"/>
      <Prop Name="FinalJobExecTimeMean" Value="0"/>
      <Prop Name="JobLengthMean" Value="617687.604"/>
      <Prop Name="CheckpointNoMean" Value="0"/>
      <Prop Name="FailedJobsLengthMean" Value="0"/>
      <Prop Name="RescheduleNo" Value="0"/>
      <Prop Name="CancelNo" Value="0"/>
      <Prop Name="SystemAvailability" Value="100"/>
      <Prop Name="SubmitInstrNo" Value="4323751862.239"/>
      <Prop Name="TotalProcessedInstr" Value="4315162273"/>
      <Prop Name="TotalProcessedInstrNoPr" Value="99"/>
      <Prop Name="UsefulProcessedInstr" Value="4315165605"/>
      <Prop Name="UsefulFinalProcInstr" Value="4315165605"/>
      <Prop Name="TotalNonIdleTimePr" Value="44.278"/>
      <Prop Name="TotalFullyLoadedTimePr" Value="44.278"/>
      <Prop Name="TotalSystemLoadPr" Value="44.278"/>
      <Prop Name="SubmittedJob" Value="6991"/>
      <Prop Name="SubmittedJobNoDouble" Value="6991"/>
      <Prop Name="JobToSchedSubmitStableState" Value="9724"/>
      <Prop Name="JobToCRSubmitStableState" Value="6991"/>
      <Prop Name="JobRejectedStableState" Value="2733"/>
      <Prop Name="CPULoadAcceptedJobsMean" Value="8.518"/>
    </Props>
   </Description>
  </Event>
 </Events>
</DSiDE>
```

An XML parser can be used to automatically extract data from this output file. Outputs of individual simulation are, for instance, utilized by DMExec to calculate final experiment results.

References

[1] Wolverine software. *GPSS and SLX Homepage*. http://www.wolverinesoftware.com/.

[2] T.J. Schriber, S. Cox, J.O. Henriksen, P. Lorenz, J. Reitman, and I. Ståhl. *GPSS Turns 40: Selected Perspectives*. In Proceedings of the 33nd conference on Winter simulation (WSC '01), Arlington, VA, December 9 – 12 2001.

[3] J. Reitman, D. Ingerman, J. Katzke, J. Shapiro, K. Simon, and B. Smith. *A Complete Interactive Simulation Environment GPSS/360-NORDEN*. In Proceedings of the 39th conference on Winter simulation (WSC '07), Washington, D.C., December 9 – 12 2007.

[4] J.O. Henriksen. *SLX: the X is for extensibility*. In Proceedings of the 32nd conference on Winter simulation (WSC '00), Orlando, FL, December 10 – 13 2000.

[5] R. Buyya and M. Murshed. *GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing*. The Journal of Concurrency and Computation: Practice and Experience (CCPE), 14(13–15), November – December 2002. Wiley Press.

[6] A. Caminero, A. Sulistio, C. Caminero, and R. Buyya. *Extending GridSim with an Architecture for Failure Detection*. In Proceedings of the 13th International Conference on Parallel and Distributed Systems (ICPADS '07), Hsinchu, Taiwan, December 5 – 7 2007.

[7] H. Casanova, A. Legrand, and M. Quinson. *SimGrid: a Generic Framework for Large-Scale Distributed Experiments*. In Proceedings of the 10th IEEE International Conference on Computer Modeling and Simulation, Cambridge, England, April 1 – 3 2008.

[8] P. Thysebaert, B. Volckaert, F. De Turck, B. Dhoedt, and P. Demeester. *Evaluation of Grid Scheduling Strategies through NSGrid: a Network-Aware Grid Simulator*. Special issue on grid computing of the Journal of Neural, Parallel and Scientific Computations, 12(3):353–378, 2004.

[9] F. Xhafa, L. Barolli, and D. Martos. *A web interface for the HyperSim-G Grid simulation package*. International Journal of Web and Grid Services, 5(1):17 – 29, March 2009.

[10] K. Ranganathan and I. Foster. *Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids*. Journal of Grid Computing, 1(1):53 – 62, 2003.

[11] L. Deboosere, M. Chtepen, and B. Dhoedt. *A Computational Load Limit-Based Algorithm for Thin Clients*. In preparation.

[12] E. Brockmeyer, H.L. Halstrøm, and A. Jensen. *The life and works of A.K.Erlang*, volume 2. Danish Acad. Techn. Sci., Köbenhavn, Denmark, 1948.

[13] F.P. Kelly. *Charging and Rate Control for Elastic Traffic*. European Transactions on Telecommunications, 8:33–37, 1997.

[14] H. Casanova and L. Marchal. *A Network Model for Simulation of Grid Application*. Technical report, École Normale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme, 2002.

3

Dynamic Scheduling of Computationally Intensive Applications on Unreliable Infrastructures

M. Chtepen, F.H.A. Claeys, B. Dhoedt, F. De Turck, P. Demeester and P.A. Vanrolleghem

Published in *Proceedings of the 2006 European Modeling and Simulation Symposium* (EMSS), Barcelona, Spain, October 4 – 6 2006.

⋆ ⋆ ⋆

In this chapter, we consider the grid distributed computing paradigm. Typical for grids is that their resources are highly distributed and often belong to different organizations. Therefore, resource availability and stability form an important issue that should be taken into account when designing scheduling mechanisms for grid systems. Currently, most existing grids make use of static schedulers, which are not sufficient to deal with the dynamic nature of grid resources. This paper is touching on the dynamic scheduling concept, justifying its usefulness for computationally intensive applications (jobs).

All results claimed in this chapter are based upon simulations done in the DSiDE simulation environment. As a use-case for this study, a computationally

intensive generic modeling and simulation tool, named Tornado, was chosen. The effect of dynamic resource availability on the execution of Tornado jobs will be shown.

3.1 Introduction

Grid computing [1] is a relatively new technology in the domain of distributed computing, which has recently gained importance as a consequence of a continuously increasing demand for elaborate sources of computational power. Grids are defined as an aggregation of heterogeneous, (globally) distributed resources, often belonging to different administrative domains. Although functionally different classes of grids exist (compute, information, desktop, *etc*.), most of them define the same set of services: through a User Interface users are able to submit their applications to a grid system; a Scheduler assigns applications received from UIs to distributed Computational and/or Storage Resources; Information Services, collects information about the grid status that helps the Scheduler to take decisions for job assignment.

The ultimate goal of grid technology is to allow for transparent execution of a users jobs, optimally exploiting the combined capacities of multiple resources while taking into account administrative policies, user requirements and system status. The ability to accomplish this goal implies the existence of an intelligent and flexible scheduling mechanism for grids.

In general, scheduling algorithms can be divided into two broad categories: static and dynamic. Static algorithms assign jobs to available resources before the execution of the jobs starts. Once jobs are running, they can no longer be interrupted by the scheduler. In case of dynamic algorithms, previously taken scheduling decisions are regularly re-evaluated and adjusted to the changing status of grid resources and jobs. It is clear that dynamic scheduling is much more complex to accomplish than its static equivalent, therefore currently there exist very few systems supporting rescheduling [2–4].

In distributed environments, such as grids, with highly dynamic resources and diverse user requirements, situations where static scheduling does not suffice to guarantee efficient workload execution often occur. For example, resources can fail or become unavailable; new resources can join the system; load on computational or network resources can vary significantly; applications with high priority or critical deadlines can arrive, requiring the best possible execution service. In grids that make use of cycle scavenging, applications are typically submitted to a number of free machines, but during execution the load of the computational resources can change drastically. At that moment the initial assignment of the workload is no longer “optimal”. In a system that employs static scheduling the jobs will continue executing on the slow machines while more appropriate resources

can be unutilized. From this example it is clear that static methods are not suitable to guarantee the optimal resource utilization in highly dynamic systems, which justifies the efforts required to develop dynamic scheduling solutions.

In this work we consider varying resource availability, *i.e.* resources becoming temporarily unavailable for job processing, and its effect on the execution of varying types of workload. We give a precise indication on the amount of jobs lost in these dynamic environments, due to job interruption in absence of a rescheduling mechanism. The remainder of this chapter is structured as follows: the assumed grid environment is described in Section 3.2; in Section 3.3 the results of simulation experiments with varying jobs executed on our unreliable grid infrastructure are presented; a case study on Tornado follows in Section 3.4; and the chapter is terminated with concluding remarks in Section 3.5.

3.2 Grid Infrastructure

A desktop grid model consisting of 10 CRs was modeled in DSiDE. Each resource is assumed to process one job at a time and has a constant speed $MIPS_{CR} = 1$. The underlying network is modeled with fixed bandwidth of 100 Mbit/s and latency of 10 ms per data transfer.

The concept of a desktop grid is that PCs of individual users are utilized for job execution during idle periods (during a lunch break, meeting, at night). However, when the owner of the machine starts his/her own applications, all external jobs have to be terminated. This dynamic resource behaviour was simulated for a varying frequency and time span of resource unavailability periods. Measurements were performed for a total grid unavailability fluctuating from 0% up to 97%. This implies that desktop resources with high utilization (where only 3% of the time can be spent on external job execution) were considered, next to systems fully dedicated to the execution of the latter. A constant arrival stream of workload was modeled with job submission times uniformly distributed from 1 to 5 minutes. Furthermore, 3 classes of jobs where considered: short (10 minutes), medium (1.5 hours) and long (3.5 hours). The grid behaviour was observed during 5 weeks of simulated time.

3.3 Simulation Results

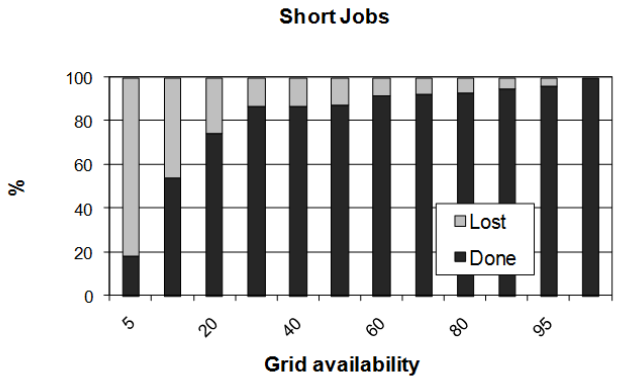
All DSiDE simulations discussed in this and the following section were executed on the UGent Grid infrastructure [5], consisting of 41 HP DL145 Dual Opteron nodes with 4 GB RAM and 40GB HD space each. All nodes are running Scientific Linux 3.0.5 and version 2.7.0 of the LCG grid middleware [6]. Only coarse-grained gridification of DSiDE experiments was performed (i.e. a simulation was

Availability	Failure	Restart
100%	U(1 s, 500 s)	U(1 s, 40 s)
95%	U(1 s, 400 s)	U(1 s, 60 s)
80%	U(1 s, 300 s)	U(1 s, 100 s)
70%	U(1 s, 300 s)	U(1 s, 150 s)
60%	U(1 s, 300 s)	U(1 s, 200 s)
50%	U(1 s, 200 s)	U(1 s, 200 s)
30%	U(1 s, 200 s)	U(1 s, 280 s)
20%	U(1 s, 200 s)	U(1 s, 350 s)
15%	U(1 s, 100 s)	U(1 s, 350 s)
10%	U(1 s, 50 s)	U(1 s, 350 s)
5%	U(1 s, 20 s)	U(1 s, 350 s)

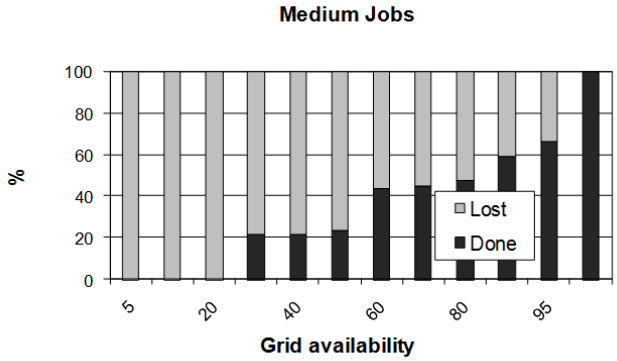
Table 3.1: Simulated unavailability and restart frequencies of grid resources.

considered as an undividable unit of work). Fine-grained gridification would entail splitting up a simulation into constituents, which is a complex task that is believed to cause severe overhead. The time to complete each simulation experiment varied from 15 minutes to 3 hours.

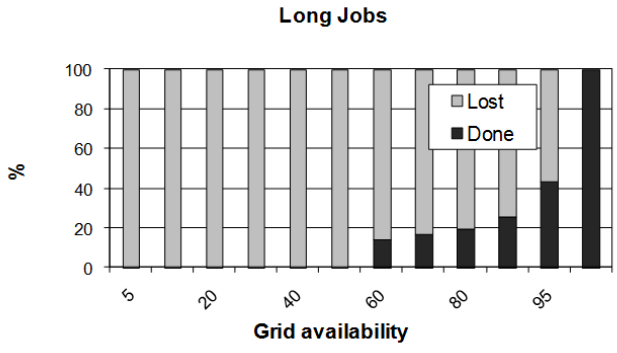
Figure 3.1 summarizes the outcomes of the experiments performed. The figure depicts the ratio of useful work processed by the modeled grid environment versus the work lost due to abrupt resource unavailability. It is important to notice that these outcomes are strongly dependent on the way a certain unavailability percentage was acquired. More specifically, the inaccessibility ratio of a resource depends on two parameters: frequency of "failure" and the time it takes the resource to "restore". Thus, the same ratio can be achieved in different ways: a resource can frequently become unavailable for a short time period; or it can become unavailable less frequently and restore its activity over a longer time interval. Throughout all the simulations described in this paper, values of resource "failure" and "restore" parameters are distributed uniformly in the intervals shown in Table 3.1. From the results in Figure 3.1, it is clear that in a heavily loaded system, the longest jobs suffer the most from system instability. When resource unavailability periods occur rarely (5% of total system time), only 3% of the work will be lost in case of short jobs, 34% in case of medium jobs and 57% in case of long jobs. Further, the frequency of a resource unavailability has a larger effect on grid performance than the time it takes a resource to restore. This can be seen from fast changing ratios between useful and lost work, which occur at points where "failure" frequency changes. More specifically, in case of medium jobs there is a sudden decrease in the percentage of useful work done when the move is made from a system that is unavailable 36% of its total run time to a grid that is unavailable half of its total run



(a)



(b)



(c)

Figure 3.1: Simulated grid performance results for different job classes: (a) short jobs, (b) medium jobs and (c) long jobs.

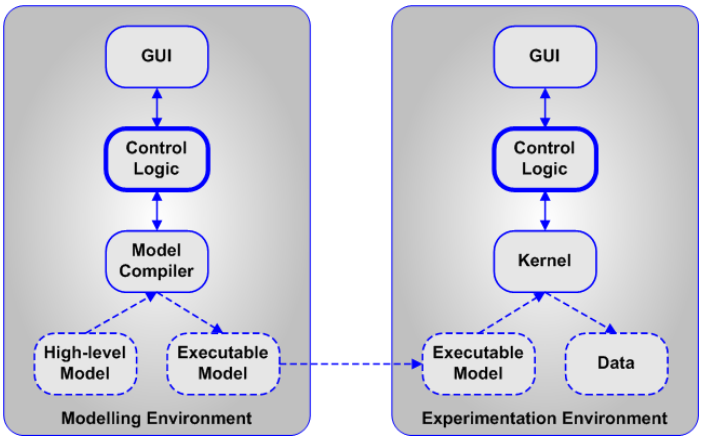


Figure 3.2: Tornado conceptual diagram.

time. At the same time, the move from 51% to 61% failure time seems to be less important. This can be explained by the fact that in the first case the failure frequency changes, while the restart frequency remains the same, and in the second case vice versa. The "jump" is larger when the job size increases.

With this simple simulation scenario, a quantitative comparison of grid performance degradation due to dynamic changing resource availability for different job classes is given. From the collected results can be concluded that a scheduler with dynamic detection of changes in resource status, combined with either job restart or migration mechanisms, can significantly improve system performance.

3.4 Case Study: Tornado Application

The aim of this PhD research is to develop efficient application-aware dynamic scheduling algorithms. As a use-case for this project, the Tornado kernel [7] was chosen. Tornado is an advanced software system for modeling and virtual experimentation with biological systems, which thus far has mainly been applied to water quality processes. Tornado has an object-oriented design and is composed of strictly separated modeling and virtual experimentation environments (see Figure 3.2).

The main elements of the Tornado modeling environment are model compiler and model builder. The model compiler converts models described in a high-level, declarative, object-oriented modeling language to flattened, executable model code. The model builder then compiles and links the executable model code into a binary object that can be dynamically loaded into the experimentation

Software	**1 run**	**1,084 runs**
Tornado	6 min	4.5 days
Tornado + LCG–2	6 min	3.5 hours

Table 3.2: Execution times for *Marselisborg* on the UGent grid infrastructure consisting of 41 HP DL145 Dual Opteron nodes

environment.

In the Tornado experimentation environment different types of virtual experiments can be designed, such as simulations, optimizations, sensitivity and scenario analyses, which are run using the executable models developed in the modeling environment. Virtual experiments can be subdivided in two categories: atomic and hierarchically structured (i.e. consisting of one or more sub-experiments), where in the second case the order of execution can be random or fixed.

Depending on model complexity, simulation intervals and desired accuracy, the execution time of a Tornado experiment varies from several minutes to several days. Therefore it seemed desirable to provide a means for distributed execution of virtual experiments. So far, only coarse-grained gridification was considered, with a simulation experiment as the smallest unit of work.

Currently, Tornado supports semi-automated execution of its jobs in two distributed environments: Typhoon [8] and LCG-2. Table 3.2 illustrates the performance improvement gained by an introduction of distributed execution. As an example the scenario analysis experiment for the Marselisborg Waste Water Treatment Plant (WWTP) in Denmark was used. The experiment was performed on the UGent Grid infrastructure, mentioned in the previous section. Distributed execution can significantly speed up the execution of Tornado jobs, however practical experience has shown that there is still a need for more advanced forms of scheduling. From September 2005 until April 2006, measurements of resource availability in the Belgian compute/data grid infrastructure for research (BEGrid) [9] were performed. The measurements suggest that the mean availability of the grids computational resources is about 85% of the observed time, and "failures" occur with varying frequency, depending on the resource provider. Since BEgrid and UGent Grid (which is a part of the BEgrid grid) both only support static scheduling of workload, all running jobs are lost, even without owner notification, each time a resource failure occurs. To measure the exact impact of such failures on execution of Tornado experiments, a number of simulations with DSiDE were performed.

The UGent Grid infrastructure is modeled consisting of 41 CRs, each able to run 2 jobs simultaneously. The speed of all resources is set to 2 MIPS and bandwidth of the network is 100 Mbit/sec. Further, the model defines a single Scheduler, running the FCFS scheduling algorithm, and a single IS. This grid model arrangement agrees with the actual architecture of the UGent Grid.

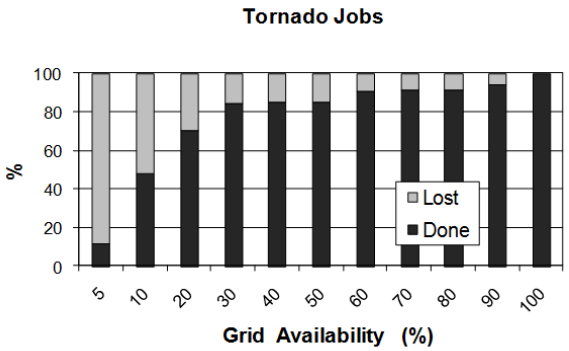


Figure 3.3: Simulated grid performance results for Tornado jobs.

Typical to the behaviour of Tornado users is that jobs are submitted once or twice a day in batches of approximately 1,000 jobs. The most common type of Tornado jobs is modeled, where job lengths are normally distributed with an average of 10 minutes and a standard deviation of a couple of minutes. Jobs use inputs of 2 Mbytes and produce 10 Mbytes output data. Table 3.1 summarizes the failure and restart frequencies that are used to model dynamic resource behaviour. The grid is observed during 5 weeks of simulated time.

Figure 3.3 shows simulation outcomes for 10 Tornado users, generating system workload following the above- described job submission pattern. In case of computational grid resources with about 10% downtime, 7% of all workload will be lost as a consequence of system instability. If the system workload increases, a larger percentage of lost jobs is expected.

3.5 Conclusions

Functional tests and calibration form an essential part of the design of efficient dynamic scheduling algorithms for grids. A number of experiments were performed using DSiDE simulator, in order to show the effect of dynamic resource unavailability periods on the execution of various types of applications.

Resource failure is just one facet of dynamic grid behaviour. The effect of other aspects, such as changing resource load, changing job characteristics *etc.* will be taken into consideration in the following chapters of this dissertation. Further, dynamic solutions for the above-stated problems will be proposed.

References

[1] I. Foster and C. Kesselman. *Software Engineering in the UNIX/C Environment*. Morgan Kaufmann, The Grid: Blueprint for a New Computing Infrastructure, 1999.

[2] A. Roy and M. Livny. *Condor and Preemptive Resume Scheduling*. Kluwer Academic Publishers, 2003.

[3] A. Barak and O. La'adan. *The MOSIX Multicomputer Operating System for High Performance Cluster Computing*. Journal of Future Generation Computer Systems, 13(4–5):361–372, March 1998.

[4] J. Gehring and A. Reinefeld. *MARS - A Framework for Minimizing the Job Execution Time in a Metacomputing Environment*. Future Generation Computer Systems, 12(1):87–99, 1996.

[5] UGent. *BeGrid UGent*. http://begrid.atlantis.ugent.be/.

[6] LHC Computing Project. *LCG (LHC Computing Grid) Website*. http://lcg.web.cern.ch/LCG/.

[7] F. Claeys, D.J.W. De Pauw, L. Benedetti, I. Nopens, and P.A. Vanrolleghem. *Tornado: a Versatile and Efficient Modelling & Virtual Experimentation Kernel for Water Quality Systems Applications*. In Proceedings of the International Environmental Modelling and Software Conference (iEMSs), Burlington, Vermont, USA, July 9–13 2006.

[8] F. Claeys, M. Chtepen, L. Benedetti, B. Dhoedt, and P.A. Vanrolleghem. *Distributed Virtual Experiments in Water Quality Management*. Journal of Hydroinformatics, 2005. Recommended.

[9] BELNET. *BeGrid (BELNET Grid Initiative) Website*. http://www.begrid.be/.

4

Adaptive task checkpointing and replication: Towards efficient fault-tolerant grids

M. Chtepen, F.H.A. Claeys, B. Dhoedt, F. De Turck, P. Demeester and P.A. Vanrolleghem

1

Published in *IEEE Transactions on Parallel and Distributed Systems*, 20(2): 180-190, 2009.

⋆ ⋆ ⋆

A grid is a distributed computational and storage environment often composed of heterogeneous, autonomously managed subsystems. As a result, varying resource availability becomes commonplace, often resulting in loss and delay of executing jobs. To ensure good grid performance, fault-tolerance should be taken into account. Commonly utilized techniques for providing fault-tolerance in distributed systems are periodic job checkpointing and replication. While very robust, both techniques can delay job execution if inappropriate checkpointing intervals and replica numbers are chosen. In this chapter we introduce several heuristics that dynamically adapt the above-mentioned parameters, based on information on grid status, to provide high job throughput in the presence of failure while reducing the system overhead. Furthermore, a novel fault-tolerant algorithm combining

checkpointing and replication is presented. The proposed methods are evaluated in the DSiDE simulation environment. Simulations are run employing workload and system parameters derived from logs that were collected from several large scale parallel production systems. Experiments have shown that adaptive approaches can considerably improve system performance, while the preference for one of the solutions depends on particular system characteristics, such as load, job submission patterns and failure frequency.

4.1 Introduction

Compared to other distributed environments, such as clusters, complexity of grids mainly originates from decentralized management and resource heterogeneity. The latter refers to hardware as well as to foreseen utilization. These characteristics often lead to strong variations in grid availability, which in particular depends on resource and network failure rates, administrative policies and fluctuations in system load. Apparently, run-time changes in system availability can significantly affect application execution. Since for a large group of time-critical or time-consuming jobs delay and loss are not acceptable, fault-tolerance should be taken into account.

Providing fault-tolerance in a distributed environment, while optimizing resource utilization and job execution times, is a challenging task. To accomplish it, two techniques are often applied: job checkpointing and job replication. In this chapter it is argued that both techniques in their pure static form are not able to cope with unexpected load and failure conditions within grids. Therefore, several solutions are proposed that dynamically adapt the checkpointing frequency and the number of replicas as a reaction on changing system properties (number of active resources and resource failure frequency). Furthermore, a novel hybrid scheduling approach is introduced that switches at run-time between checkpointing and replication depending on the system load. Decisions taken by the above-mentioned algorithms are primarily based on monitored grid state, but also on job characteristics and on collected historical information. Currently, the proposed techniques are limited to address hardware failure in grids running applications composed of independent jobs.

Simulation-based experiments, using the DSiDE grid simulator and a data set derived from realworld logs collected from different large scale parallel production systems [1, 2], have shown that the adaptive approaches significantly improve distributed system performance. They achieve throughput and fault-tolerance comparable with that of static checkpointing and replication with optimal parameters. However, to make an appropriate choice between strategies, some knowledge on system parameters is still required. To deal with the latter issue, the hybrid approach, combining the advantages of both techniques, may be preferred.

The remainder of this chapter is organized as follows: Section 4.2 discusses

related work; Sections 4.3 elaborate on adaptive checkpointing and provides a simulation-based comparison between different checkpointing approaches; Section 4.4, in turn, discusses and compares replication-based and hybrid sheduling solutions; while section 4.5 concludes this discussion.

4.2 Related Work

A large number of research efforts have already been devoted to fault-tolerance in the scope of distributed environments. Aspects that have been explored include the design and implementation of fault detection services [3, 4], as well as the development of failure prediction [2, 5–7] and recovery strategies [8–10]. The latter are often implemented through job checkpointing in combination with migration and job replication. Although both methods aim to improve system performance in the presence of failure, their effectiveness largely depends on tuning run-time parameters, such as the checkpointing interval and the number of replicas [11–13]. Determining optimal values for these parameters is far from trivial, for it requires good knowledge of the application and the distributed system at hand.

4.2.1 Checkpointing

To tackle the checkpointing overhead and scalability concerns, different approaches are addressed in literature. One well researched technique is known as incremental checkpointing [14]. It reduces data stored during checkpointing to only blocks of memory modified since the last checkpoint. In [15] a checkpointing-based fault tolerance protocol for MPI jobs is presented, which lowers the overhead during normal execution and allows fast crash recovery by using the ideas of message logging and object-based processor virtualization. The latter limits the re-execution to only the failed processor and allows to distribute the failed work among the other processors. Clearly, this approach is only applicable to homogeneous environments. Yet another important approach is based on determination of the optimal checkpointing frequency and is called the optimal checkpoint interval problem. Several researches have addressed this problem [16–18], but they have provided analytical solutions applicable only under specific system assumptions. For instance, it is often assumed that interoccurrence times of failures and repairs for each resource are independent and exponentially distributed. In practice failures tend to cluster in time, while being caused by a relatively small set of computational nodes [2, 6, 7]. Since optimal solutions do not appear to be generally applicable, static sub-optimal solutions were addressed. For instance, in [19] a min-max checkpoint placement method is introduced that determines the sub-optimal checkpoint sequence under uncertain circumstances in terms of the system failure time distribution. However, even if the (sub)optimal checkpointing

interval is computed beforehand, the distributed system or application parameters upon which the interval is based will presumably change over time. Therefore, new forms of checkpointing optimization were recently considered in literature. One of them is the so-called cooperative checkpointing concept, introduced in [20, 21], which addresses system performance and robustness issues by allowing the application programmer, the compiler and the run-time system to jointly decide on the necessity of each checkpoint. The checkpointing algorithms proposed in this paper are based on this concept and thus are cooperative (adaptive) heuristics. In [22] another set of cooperative checkpointing schemes is proposed that dynamically adjust the checkpointing interval with as an objective timely job completion in the presence of failure. The schemes use information on remaining job execution time, time left before the deadline and the expected remaining number of failures before job termination. The latter implies that the system failure distribution should be known in advance. [23], in turn, considers only dynamic checkpointing interval reduction in case it leads to computational gain, which is quantified by the sum of the differences between the means for fault-affected and fault-unaffected job response times. In [24] yet another adaptive fault management scheme (FT-Pro) is discussed. Opposite to the combined approach proposed in this paper that uses adaptive checkpointing in combination with replication, FT-Pro combines adaptive checkpointing with proactive process migration. The approach optimizes application execution time by considering the failure impact and the prevention costs. FT-Pro supports three prevention actions: skip checkpoint, take checkpoint and migrate. The appropriate action is selected based on the predicted frequency of failure. Therefore, the effectiveness of FT-Pro strongly depends on the quality of this prediction.

4.2.2 Replication

Similar to deciding upon the best checkpointing interval, finding a generally applicable procedure to calculate the optimal number of job replicas is a complicated issue. Several studies have attempted to address this problem [12, 25], but unfortunately they enforce a number of restrictions on the execution environment, job interdependency, *etc*. Nowadays, most of the replication-based fault-tolerant algorithms assume a fixed number of job duplicates. However, dynamic solutions have recently started to receive attention. In [9] a dynamic replication-based method is described, called Workqueue with Replication (WQR). Initially, the algorithm distributes a single copy of a job to random idle resources in FCFS order. When the job queue is empty and the system has free resources, replication is activated to cope with varying availability of hosts. The disadvantage of this "delayed-copy" approach is that if a system is heavily loaded for a long period, which is often the case in large scientific or production grids, the replication will be significantly

delayed or not activated at all. Furthermore, as was mentioned in [6], most of the failures in distributed environments tend to occur during peak hours, when the WQR failure prevention is turned off by definition. Other interesting research on job replication is reported on in [26] where a group-based dynamic replication mechanism for peer-to-peer grid computing environments is proposed. Whereas the algorithms introduced in our paper dynamically vary the number of job replicas dependent on the system load, the group-based approach determines the amount of replication taking into account the reliability of each volunteer group, which is a group of resources with similar properties.

4.2.3 Combined approaches

Several papers [27, 28] describe schemes that combine checkpointing and job replication to deal with transient fault detection. Transient faults are often hard to detect because they do not result in a resource crash but only in a job state modification, which however can lead to wrong output. Therefore, duplicate jobs are executed on different nodes and their state is compared to track faults. The checkpointing mechanism, in turn, serves two purposes: preservation of a job state, to reduce the fault-recovery time; and state comparison of job replicas. To our knowledge, no work combining checkpointing and replication was performed thus far with the objective of achieving better resource utilization and improving job execution time.

4.3 Adaptive Checkpointing Heuristics

4.3.1 The checkpointing model

The grid model considered in this paper consists of geographically dispersed computational Sites (S), aggregating altogether 128 Computational Resources and a number of general services (Figure 4.1). The latter include a User Interface (UI), through which jobs are submitted into the system; a Grid Scheduler (GS) responsible for job-resource matchmaking; an Information Service (IS), which collects job and resource status information required by the GS; and a Checkpoint Server (CS) where checkpointing data is made persistent. The GS invokes the matchmaking procedure within the predefined scheduling interval I_{GS}, while the IS collects changes in resource status with a delay I_{IS}, to reflect the modification propagation time occurring in actual deployments. The grid sites reside within a WAN, while resources belonging to a single site are interconnected by LANs. Finally, it is assumed that all grid management services are protected against failure and only CRs are unstable, with a resource failure affecting all CPUs within a CR. Contrary to the traditional assumptions considering failures to be independent and equally spread over all system resources with a particular distribution, failures in this work

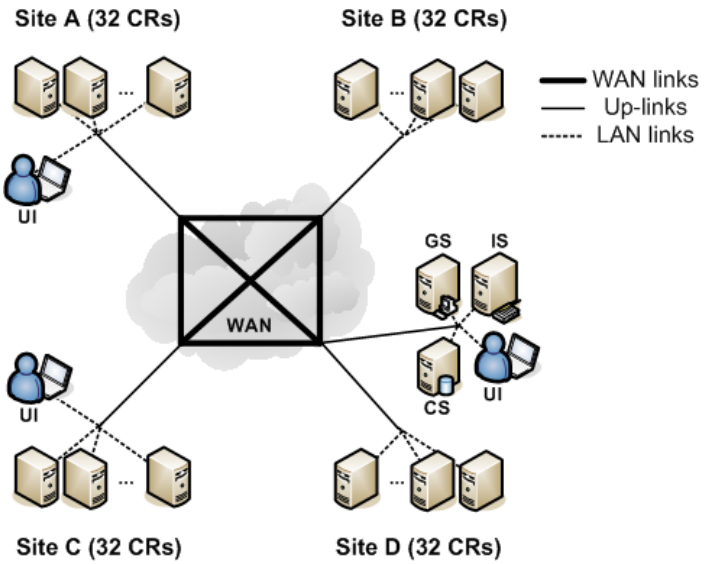


Figure 4.1: Example grid architecture: User Interface (UI), Grid Scheduler (GS), Information Service (IS), Checkpoint Server (CS), Wide Area Network (WAN), Local Area Network (LAN).

can be spatially and temporarily correlated, which has proven to be a more realistic presumption in case of large scale distributed systems [2, 6, 7].

In this model, the benefits of checkpointing are limited by the following factors: the run-time overhead (C), which is the time delay resulting from interruption of job execution to perform checkpointing; the network latency (L) (a time interval between the checkpoint generation and its availability on the CS); and the recovery delay (R), which is the time to download a failed job checkpoint from the CS to the CR where the job is rescheduled to run. The L and R parameters are mainly determined by the available network capacities, the distance between the CS and the considered resource and checkpoint size. The values of both parameters could be reduced by applying checkpoint replication on multiple storage servers. However, this paper concentrates on the reduction of the checkpointing run-time overhead and therefore proposes several algorithms that differentiate the checkpointing interval (I) based on history statistics and current status of a particular job and its execution environment. By this means we will on the one hand eliminate unnecessary checkpointing and on the other hand introduce extra job state savings where the danger of failure is considered to be severe. More specifically, the optimal checkpointing interval for a job J (I_J^{opt}) running on the computational node CR depends on the following parameters: E_J^{CR} is the execution time of J on the resource CR (taking into account load of CR); F_{CR} is the

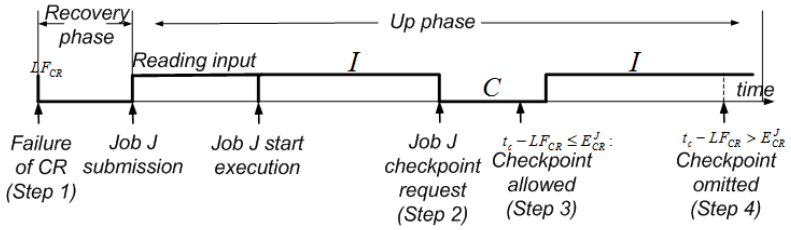


Figure 4.2: Operation of LastFailureCP on a resource running a single job.

average time between failures of CR; and CS^J is the size of the J checkpoint. Additionally, the value of I_J^{opt} should satisfy the inequality $C < I_J^{min} \leq I_J^{opt}$ to be sure that jobs make execution progress despite of periodic checkpointing. I_J^{min} is the minimum checkpointing interval of J, which should be initialized with a default value, for example a small percentage of E_J^{CR}. It is considered that after the I_J^{CR} interval expires, either the next checkpointing event can immediately be performed by the application, or a flag is set indicating that the checkpointing can be accomplished as soon as the application is able to provide a consistent checkpoint. Furthermore, it is important to notice that deciding upon the job execution time is a complicated problem, often requiring an application-specific approach [29]. In our paper the problem is simplified by assuming that the execution time can be exactly determined in advance. Therefore, the simulation results presented in the following sections show the upper bounds of the algorithms performance, with respect to this parameter. In the next sections the proposed algorithms are discussed in more detail.

4.3.2 Last Failure Dependent Checkpointing (LastFailureCP)

The main disadvantage of unconditional periodic job checkpointing (PeriodicCP) is that it performs identically whether the job is executed on a volatile or on a stable resource. The goal of LastFailureCP is to reduce the overhead introduced by excessive checkpointing in relatively stable distributed environments, *i.e.*, the algorithm omits unnecessary checkpoints of the job J based on its estimated total execution time and the failure frequency of the resource CR, to which J is assigned (see Figure 4.2). For each resource the algorithm keeps a timestamp LF_{CR} of its last detected failure (Step 1). When no failure has occurred, LF_{CR} is initiated with the system start time. After an execution interval I, each job running on an active resource generates a checkpointing request (Step 2). The request is subsequently evaluated by the GS and it is allowed only if the comparison $t_c - LF_{CR} \leq E_J^{CR}$ evaluates to true (Step 3), where t_c is the current system time. As was previously mentioned, each checkpoint generation leads to run-time overhead C, which prolongs the execution of J (Step 3). If $t_c - LF_{CR} > E_J^{CR}$, the checkpoint is omitted

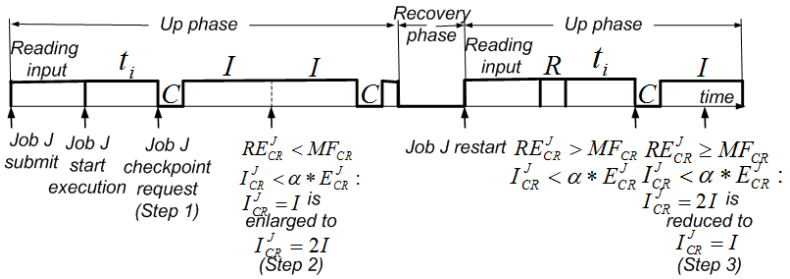


Figure 4.3: Operation of MeanFailureCP on a resource running a single job.

to avoid the overhead as it is assumed that the resource is "stable" (Step 4). To prevent excessively long checkpoint suspension, a maximum number of omissions can be defined.

4.3.3 Mean Failure Dependent Checkpointing (MeanFailureCP)

Contrary to LastFailureCP that only considers checkpoint omissions, MeanFailureCP dynamically modifies the initially specified checkpointing frequency to deal with inappropriate checkpointing intervals (see Figure 4.3). The algorithm modifies the checkpointing interval based on the run-time information on the remaining job execution time (RE_J^{CR}) and the average failure interval (MF_{CR}) of the resource CR where the job J is assigned, which results in a customized checkpointing interval I_J^{CR}. Use of MF_{CR}, instead of LF_{CR}, reduces the effect of an individual failure event. While PeriodicCP and LastFailureCP are first run after the expiration of the predefined checkpointing interval, the MeanFailureCP activates checkpointing within a fixed and preferably short time period t_i after the beginning of a job execution (Step 1). The latter approach opens the possibility to modify the checkpointing frequency at the early stage of job processing. Each time the checkpointing is performed, I_J^{CR} is adapted as follows: if $RE_J^{CR} < MF_{CR}$ and $I_J^{CR} < \alpha \times E_J^{CR}$, where $\alpha < 1$, the frequency of checkpointing will be reduced by increasing the checkpointing interval $I_{J_{new}}^{CR} = I_{J_{old}}^{CR} + I$ (Step 2). The first inequality in the condition ensures that either CR is sufficiently stable or the job is almost finished, while the second limits the excessive growth of I_J^{CR} compared to the job length. The latter can particularly be important for short jobs, for which the first condition almost always evaluates to true. On the other hand, when the above-mentioned inequalities are not satisfied, it seems to be desirable to decrease I_J^{CR} and thus to perform checkpointing more frequently $I_{J_{new}}^{CR} = I_{J_{old}}^{CR} - I$ (Step 3). When reducing the checkpointing interval, the following constraint should be taken into account: $C < I_{min} \leq I_{J_{new}}^{CR}$. I_{min} is a predefined value, which secures that the time between consecutive checkpoints is never less than the time overhead

added by each checkpoint. In case of stable grid systems it is desirable to choose relatively large values for I_J^{min} (5% - 10% of the total job length) to prevent an undesirably steep decrease of the checkpointing interval. Experiments have shown that gradually incrementing I_J^{CR} by I ensures rapid achievement of I_J^{opt} in most distributed environments. However, in case of rather reliable grids, the calibration of I_J^{CR} can be accelerated by replacing I with a desirable percentage of the job execution time.

4.3.4 Simulation Results

Since grids are complex and often unpredictable environments, it is difficult to build a grid testbed on a realistic scale for validation and calibration of grid scheduling strategies. Therefore, the algorithms proposed in this paper were validated using the DSiDE grid simulator. In addition to standard events, such as resource and job registration, DSiDE supports several types of dynamic system modifications, including alternating resource availability. The latter is modelled as a sequence of failure and restore events occurring with particular distributions. Failure events can either be correlated or independent. Therefore, the total grid resource availability, which is the percentage of time during which the resource performed useful computations, can be defined as:

$$A_{CR} = ((1 - (\sum_{n=1}^{N}(t_{CR,n}^{f} - t_{CR,n}^{r})/T^{sim})) \times 100) \tag{4.1}$$

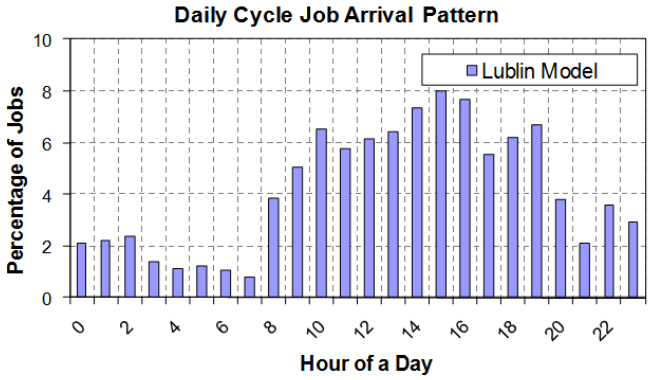
where N is the number of resource failures; $t_{CR,n}^{f}$ and $t_{CR,n}^{r}$ are respectively the timestamp of the resource failure and restore; and T^{sim} is the total wall clock simulation time. From the individual resource availability, total grid availability is computed as follows:

$$A_{grid} = (\sum_{n=1}^{N^{total}} A_{CR_n})/N^{total} \tag{4.2}$$

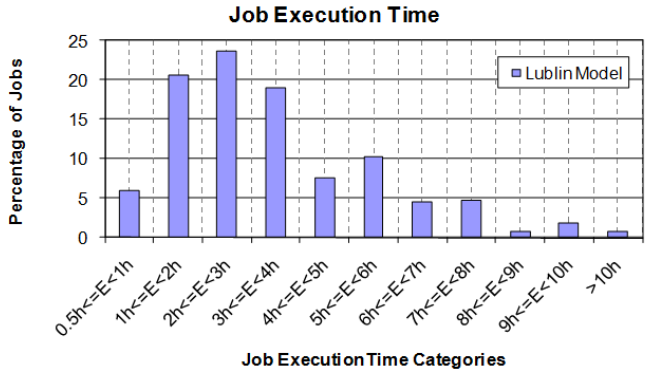
where N^{total} is the total number of resources in the grid.

It is assumed that grid services and the underlying network are fully reliable and thus failures can only originate from CRs.

To compare the performance of the proposed checkpointing heuristics, realistic workload and system failure models, derived from production grid logs, were utilized. More specifically, the submitted workload follows the Lublin job generation model [30], where execution of batch jobs running on a single node was assumed. In this simulation scenario the model parameters are initialized to represent a heavily loaded grid system with a daily cycle job arrival pattern. Figure 4.4(a) depicts the proportion of jobs submitted into the grid system each hour, during an observation period of 24 hours. Apparently, most of the jobs (almost 80%) arrive



(a)



(b)

Figure 4.4: Lublin workload model: (a) job arrival pattern with daily cycle; (b) job execution time distribution.

during the day-time, while the remaining 20% are submitted between 21 PM and 7 AM. The run-times of the submitted jobs vary as shown in Figure 4.4(b), where 11 categories of job lengths (from less than an hour to more than 10 hours) are differentiated. More than 80% of all submitted jobs have medium execution times, varying from 1 hour to 6 hours. Furthermore, to simplify the comparison between different algorithms, it is assumed that all jobs use inputs and outputs of 10 MB; and a job checkpointing delay varies from 100 ms to 5 s, depending on the execution time.

The above-described workload is submitted into the grid system discussed in Section 4.3.1. The system parameters are set as follows: WAN links have equal bandwidth of 100 Mit/s and latency varying from 3 to 10 ms; CRs inside the

sites communicate through LAN networks arranged into a star-topology, with link bandwidths of 100 Mbit/s and latency of 1 ms; GSched is run every 10 min, while the longest propagation delay for the IS is initialized to 5 min; and, finally, each CR has 1 MIPS CPU speed and is limited to process at most 2 jobs simultaneously. Failure and restore patterns of the grid resources follow the model represented in [2]. These models are constructed based on the analysis of failure data collected over the past 9 years at Los Alamos National Laboratory, which is currently one of the largest high-performance computing sites worldwide. In the grid environment considered each of the 4 sites is modelled to posses different failure and restore behaviour. Failure frequency ranges over the sites from several hours to several weeks and is modeled by a Weibull distribution with decreasing hazard rate. Mean repair time, in turn, varies across the sites from less than an hour to more than a day and is modeled by a logarithmic distribution. The considered grid system has a total availability of 90%.

The grid model described was observed during 7 days of simulated time. Figures 4.5(a) - 4.5(b) and 4.6(a) - 4.6(b) show a comparison between the performance of the proposed dynamically adapting heuristics and the PeriodicCP approach for a randomly varying initial checkpointing interval I. From the figures it is clear that the efficiency of PeriodicCP strongly depends on the chosen value of I, which remains constant during the simulation. For instance, overly frequent as well as scarce checkpointing can result in up to 40% decrease in number of processed jobs, compared to the best achieved situation, and significantly increase the average job execution time. Furthermore, from Figure 4.6(b) can be observed that at high checkpointing frequencies the average job length significantly decreases. This relates to the fact that exaggerated checkpointing substantially prolongs job execution and therefore only short jobs finish within the observed time interval. However, when I decreases and longer jobs can get processed, an increase in job run-time is in effect.

The results achieved with PeriodicCP are partially improved by LastFailureCP due to omittion of redundant checkpoints. Apparently, the technique provides the best results for short checkpointing intervals. Since the algorithm does not consider checkpoint insertion, it performs slightly worse than PeriodicCP for large values of I. However, in the latter case the effectiveness of LastFailureCP strongly depends on failure periodicity. In the best case when failures occur quite periodically and thus can easily be predicted by the algorithm, LastFailureCP will perform similar to PeriodicCP.

Finally, the fully dynamic scheme of MeanFailureCP proves to be the most effective. Starting from a random checkpointing frequency it results in a number of executed jobs and average job run-time that are close to the results achieved by PeriodicCP with the best performing checkpointing interval. Important to notice is the slight decrease in the number of checkpoints taken by MeanFailureCP as

I is getting closer to the best performing checkpointing values. This decrease can be explained by a shorter calibration period required to achieve the "optimal" value of I. On the other hand, for large values of I an increase in the number of generated checkpoints is observed, which is the consequence of the constraint $I_{min} \leq I_J^{CR} < \alpha \times E_J^{CR}$, where I_{min} and α were initialized with respectively $0.01 \times E_J^{CR}$ and 1. When I_J^{CR} grows, this restriction evaluates to false for a larger part of jobs and therefore their adapted checkpointing interval starts to decrease, resulting in more checkpoints performed. As can be seen from the simulation results, this selective increase in checkpointing keeps the number of processed jobs and the average execution time of MeanFailureCP more or less constant, while in the case of the PeriodicCP and LastFailureCP algorithms the performance drops considerably.

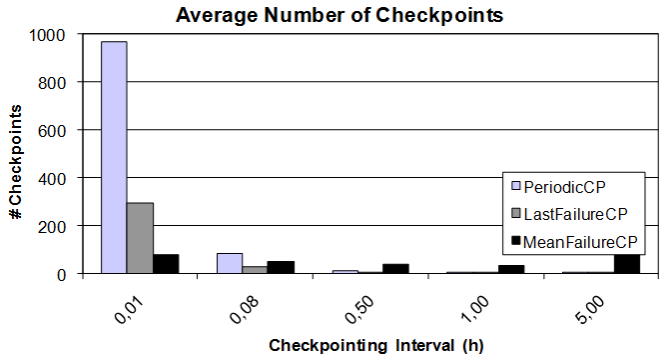
4.4 Replication-based Heuristics

4.4.1 Load Dependent Replication (LoadDependentRep)

Providing fault-tolerance in distributed environments through replication has as an advantage that otherwise idle resources can be utilized to run job copies without significantly delaying the execution of the original job. Obviously, the more job copies are running on the grid, the larger is the chance that one of them will execute successfully. On the other hand, running additional replicas on a distributed environment with an insufficient number of free resources can considerably reduce throughput and prolong job execution. To deal with this dilemma, the proposed heuristic considers the system load and postpones or reduces replication during peak hours. The algorithm requires a number of parameters to be provided in advance, *i.e.* the minimum (Rep^{min}) and maximum (Rep^{max}) number of job copies, and the CPU limit (CL). The latter parameter specifies the lower bound on the number of active free CPUs for replication to take place. An example of the heuristic operation is shown in Figure 4.7, where the required parameters are initialized as follows: Rep^{min} and Rep^{max} are set respectively to 1 and 2; and CL equals to 2 CPUs. In each iteration, the GSched consults the IS for the system status (Step 1). Based on this information, CA and CL are compared, where CA is the number of active CPUs able to execute the next job. The outcome of the comparison determines the choice for the next job to be scheduled:

- $CA \geq CL$: select a job J with the earliest arrival timestamp and the number of active replicas less than Rep^{max} (Step 1)
- $0 < CA < CL$: select a job J with the earliest arrival timestamp and the number of active replicas less than Rep^{min} (Step 2)
- $CA = 0$: skip the current scheduling round (Step 3).

(a)



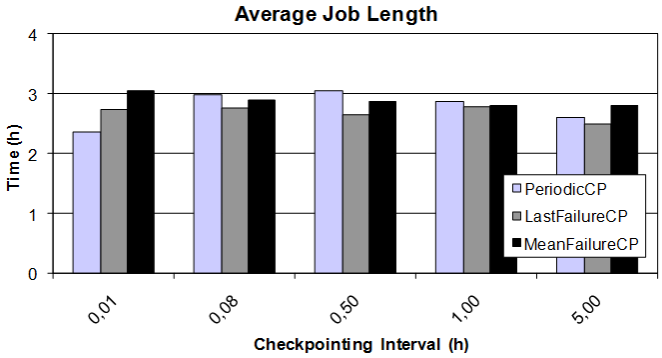
(b)

Figure 4.5: Checkpointing heuristics performance for varying initial checkpointing interval: (a) number of successfully executed jobs; (b) average number of checkpoints initiated by different heuristics.

However, even if the grid system is heavily loaded it can be desirable to consider $Rep^{min} > 1$, since the failure rate of resources in distributed environments increases with the intensity of the workload running on them. When one of the job duplicates finishes, other replicas are automatically canceled (Step 4). If the system load decreases before the job was executed, the remaining $Rep^{max} - Rep^{min}$ replicas are activated (Step 5).

The algorithm assigns the selected job J to the site S with some free resources and with the smallest number of J replicas (Step 1, Step 5), since spreading replicas over different sites increases the probability that one of them will be successfully executed. If multiple sites have an equal number of job copies, a site that can provide for the fastest job execution is preferred. The speed or capacity of a site is

(a)



(b)

Figure 4.6: Checkpointing heuristics performance for varying initial checkpointing interval: (a) job average run-time; (b) job average length.

defined as:

$$MIPS_S = (\sum_{CR \in S} MIPS_{CR})/((\sum_{CR \in S} n_{CR}) + 1) \tag{4.3}$$

where $MIPS_{CR}$ is the speed of CR and n_{CR} is the number of jobs on CR. In the above equation, only resources executing no other replicas of J are taking into account. Distribution of similar replicas to a single CR is avoided because it is assumed that CPUs inside a single node have more chance to fail simultaneously in case of the resource malfunction. Therefore, inside the chosen site the job will be submitted to the fastest available resource with no identical job replicas: $max(MIPS_{CR}/(n_{CR} + 1))$. If no such resource exists the distribution of J is postponed and the next job from the GS queue is scheduled.

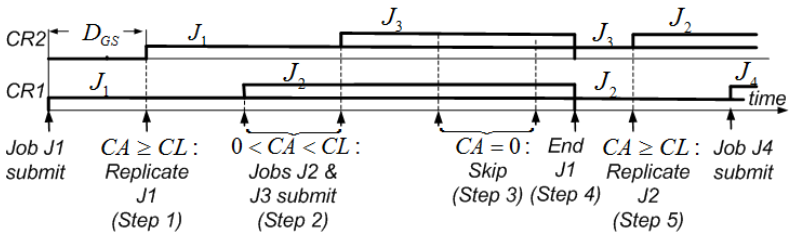


Figure 4.7: Operation of LoadDependentRep on a distributed environment consisting of 2 resources, each able to run 2 jobs simultaneously. $Rep^{max} = 2$, $Rep^{min} = 1$ and $CL = 2$.

4.4.2 Failure Detection and Load Dependent Replication (FailureDependentRep)

To increase fault-tolerance of the previously discussed LoadDependentRep heuristic, the approach was combined with a failure-detection technique. The principle of failure detection is straightforward: as soon as a resource failure is discovered by the GS, all jobs submitted to the failed resource are redistributed. The algorithm proceeds as LoadDependentRep, except that in each scheduling round not only newly arrived jobs are considered for submission to a CR, but also all jobs distributed to failed nodes. To construct a list of active CRs, GS queries the IS. However, in order for a resource failure to be detected, the restore time should exceed $I_{IS} + I_{GS}$. This means that although the method offers a higher level of fault-tolerance compared to solely replication-based strategies, it does not ensure job execution.

4.4.3 Adaptive Checkpoint and Replication-Based Fault- Tolerance (CombinedFT)

In this section a combined checkpointing and adaptive replication-based scheduling approach is considered that dynamically switches between both techniques based on run-time information on system load. The algorithm can be particularly advantageous for grids with frequent or unpredictable alternations between peak hours and idle periods. In the first case, replication overhead can be avoided by switching to checkpointing, while in the second case the checkpointing overhead is reduced by using low-cost replication. An example of the CombinedFT heuristic operation is shown in Figure 4.8, where the required parameters are initialized as follows: Rep^{min} and Rep^{max} are set respectively to 1 and 2; and CL equals to 2 CPUs.

When the CPU availability is low ($CA < CL$) the algorithm is in checkpointing mode (Step 1). In this mode, CombinedFT rolls back, if necessary, the earlier dis-

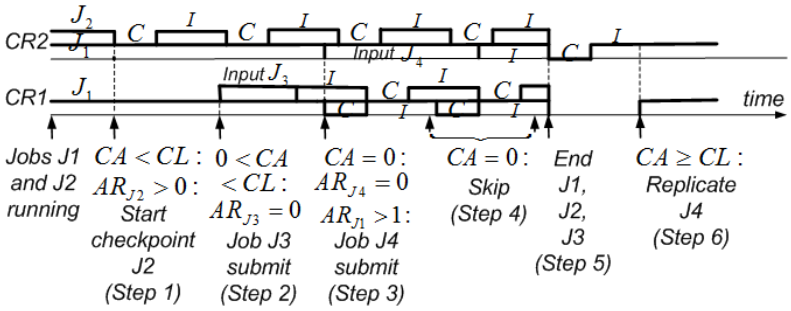
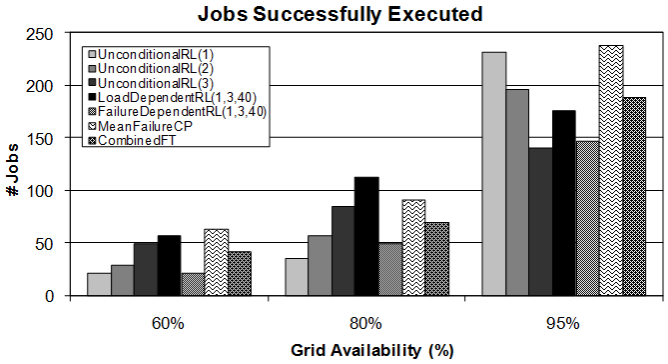


Figure 4.8: Operation of CombinedFT on a distributed environment consisting of 2 resources, each able to run 2 jobs simultaneously. $Rep^{max} = 2$, $Rep^{min} = 1$ and $CL = 2$. The PeriodicCP method is applied in the checkpointing mode.

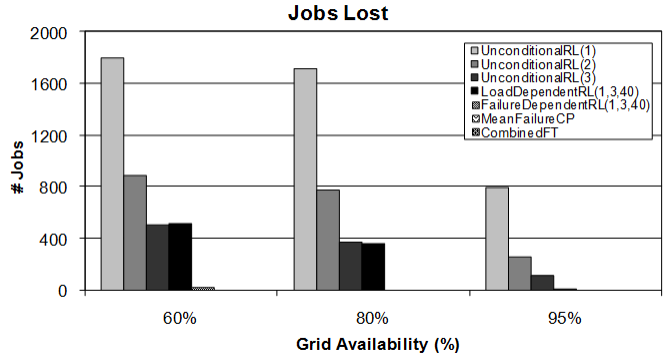
tributed active job replicas (AR_J) and starts job checkpointing. When processing the next job J, the following situations can occur:

- $AR_J > 0$: start checkpointing the most advanced active replica, cancel execution of other replicas (Step 1)
- $AR_J = 0$ **and** $CA > 0$: start J on the least loaded available resource within the least loaded site, determined respectively by (2) and (1) (Step 2). Start checkpointing of J.
- $AR_{J_1} = 0$ **and** $CA = 0$ **and** $\exists J_2 : AR_{J_2} > 1$: select a random replicated job J_2 if any, start checkpointing its most advanced active replica, cancel execution of other replicas of J_2, submit J_1 to the best available resource (Step 3)
- $AR_{J_1} = 0$ **and** $CA = 0$ **and** $\neg\exists J_2 : AR_{J_2} > 1$: skip the current scheduling round (Step 4).

The algorithm switches to replication mode when either the system load decreases or enough resources restore from failure ($CA \geq CL$) (Step 5). In replication mode all jobs with less than Rep^{max} replicas are considered for submission to the available resources, in the order defined by the FailureDependentRep algorithm (see Section 4.4.1). When a job J is selected, it is assigned to the fastest resource (with no similar job replicas) connected to a grid site S with the maximum $MIPS_S$ and the smallest number of identical replicas. If J was previously in checkpointing mode and the replication completed successfully, the checkpointing of J is switched off (Step 6).



(a)

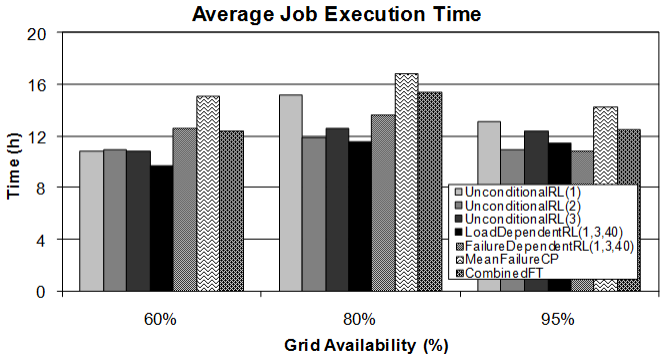


(b)

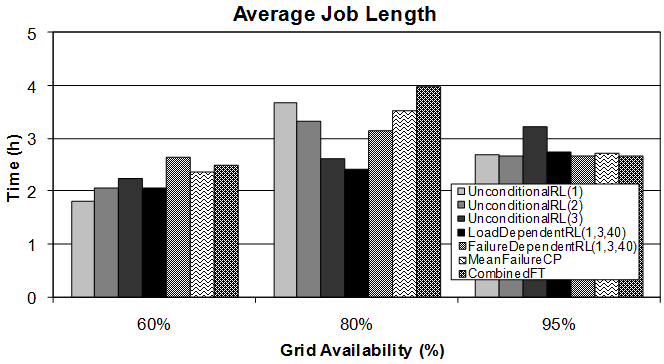
Figure 4.9: Performance of replication-based, checkpointing-based and hybrid algorithms on heavily loaded grids with varying availability: (a) number of successfully executed jobs; (b) number of jobs lost.

4.4.4 Simulation Results

In this section the performance of the replication-based and hybrid approaches is compared against the performance of the best checkpointing heuristic (MeanFailureCP). The comparison is performed within grid systems with varying load and availability. Four replication algorithms are considered: UnconditionalRep(2), unconditional job replication with 2 job copies; UnconditionalRep(3), unconditional job replication with 3 copies; LoadDependentRL(1,3,40) adaptive replication with the minimum (Rep^{min}) and maximum (Rep^{max}) number of job replicas set to respectively 1 and 3, and the free CPU limit initialized to 40 (approximately 1/3 of the total grid capacity); FailureDependentRep(1,3,40), failure detection and adaptive replication based algorithm with the same parameters as LoadDependentRep.



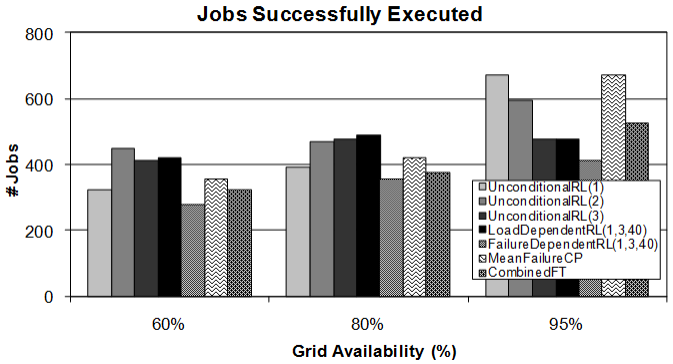
(a)



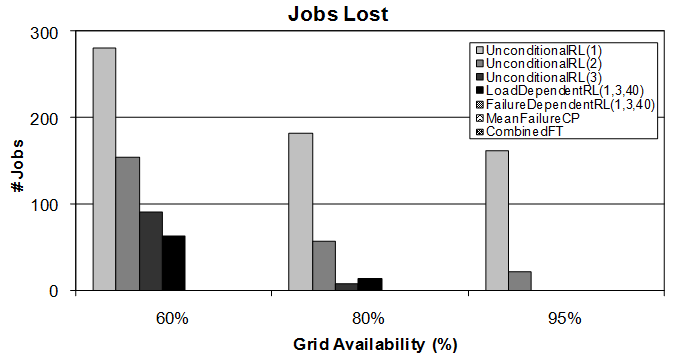
(b)

Figure 4.10: Performance of replication-based, checkpointing-based and hybrid algorithms on heavily loaded grids with varying availability: (a) average job run-time; (b) average job length.

Also the performance of FCFS (or UnconditionalRep(1)) was observed to serve as a reference for comparison with the other algorithms. The combined approach (CombinedFT) is initialized with the same replication parameters as FailureDependentRep and switches in the checkpointing mode to the MeanFailureCP approach. The chosen parameter values for the replication-based heuristics are not necessarily optimal but they are believed to be reasonable for the case at hand. The term "unconditional job replication algorithm" refers to an algorithm that sequentially processes jobs arriving to the GS, based on the timestamp of their arrival. Independent of the current system load, the algorithm creates for each job a predetermined number of replicas that are assigned to different available resources, until all resources are filled.



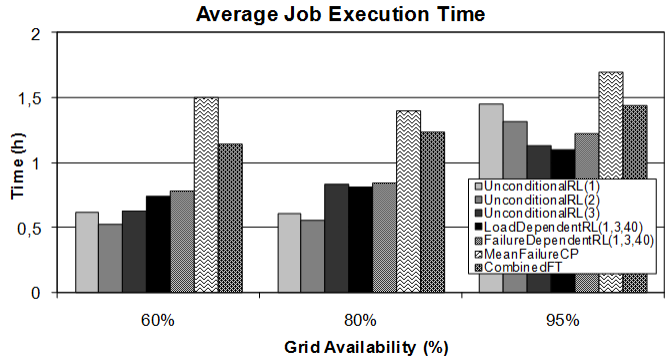
(a)



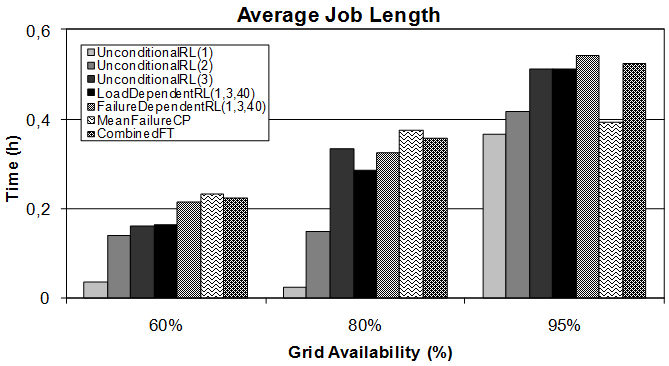
(b)

Figure 4.11: Performance of replication-based, checkpointing-based and hybrid algorithms on grids with low load: (a) number of successfully executed jobs; (b) number of jobs lost.

The algorithms are evaluated for high and low grid loads, which are accomplished by varying the job submission parameters of the Lublin model (see Section 4.3.4). In the case of high grid load, the same model parameters are utilized as the ones applied in the simulation scenario of the Section 4.3.4. In this scenario about 7,000 jobs arrive into the system during the observation period of 7 days (simulated time), which leads to long periods of system overload alternating with relatively short "idle" time-intervals. In the case of low grid load, jobs are generated occasionally (about 700 during 7 days of simulated time), while most of the time a large part of the resources remains idle. To warrant low system utilization, also the average job length is reduced from 2.5 hours in the first scenario to 0.3 hours in the latter. The size of job input and output data in both simu-



(a)



(b)

Figure 4.12: Performance of replication-based, checkpointing-based and hybrid algorithms on grids with low load: (a) average job run-time; (b) average job length.

lation scenarios is set to 10 GB, to yield large data volumes often generated by real-world computationally intensive applications. Finally, varying system availability is achieved by modifying the parameters of the Weibull distribution within Schroeder and Gibson's model (for more details see Section 4.3.4). It is again considered that each site possess different failure and restore patterns, with failure and restore intervals varying respectively from 2 hours to a week and from 30 min to a day.

Figures 4.9(a) - 4.9(b) and 4.10(a) - 4.10(b) visualize the evaluated scheduling methods' performance on a highly loaded grid, while Figure 4.11(a) - 4.11(b) and 4.12(a) - 4.12(b) summarizes the results for low grid load. The following system parameters are observed: number of successfully executed / lost jobs, average job execution time and average job length. Important to notice is that a replicated

job is assumed to be lost when all its replicas were started and afterwards failed.

For heavily as well as lightly loaded grids with relatively low availability, additional replication clearly provides better system performance and lower job loss rate. This is the consequence of the fact that replication-related overhead is compensated by increased grid reliability and consequently by a higher ratio of successfully executed jobs. However, as the grid availability improves (95%), additional replication provided by UnconditionalRep(2) and UnconditionalRep(3) leads to system throughput reduction. This reduction is getting more significant as the grid load increases and resources become more scarce. The results for LoadDependentRep show that the performance of unconditional replication can be improved by postponing the execution of additional replicas during the peak hours. The higher the system load, the more gain can be achieved from the postponement (see Figure 4.9(a)). On the other hand, for slightly loaded systems (see Figure 4.11(a)), LoadDependentRep performs only slightly better than UnconditionalRep(3), since replication almost never has to be delayed. The main advantage of FailureDependentRep is certainly its high reliability in absence of sophisticated mechanisms for providing fault-tolerance. Implementation of the algorithm requires only a replica counter and a simple job monitoring facility. Another benefit is that failure-sensitive long jobs have a higher chance to finally get processed due to the restart mechanism (see longer average job lengths in Figures 4.10(b) and 4.12(b)). The disadvantage of FailureDependentRep is the slower grid performance (larger average execution times) as a result of the postponed replication in combination with the run-time overhead related to repetitive restart of failed jobs (see Figures 4.10(a) and 4.12(a)). However, lightly loaded systems are less sensitive to this last disadvantage since most of the time enough computational resources are available and thus multiple job restarts do not penalize the execution of other jobs. In the condition of high load, the fully fault-tolerant MeanFailureCP results in the best system throughput compared to the other considered heuristics. This is the consequence of the considerable overhead introduced by the execution of additional replicas in an overloaded grid system. On the other hand, the average job execution time in case of the checkpointing approach is always relatively high, which leads to the algorithm performance reduction in the lightly loaded grid, where replication provides for almost costless fault-tolerance. However, it is important to notice that the exact relation between the performance of the checkpointing and the replication-based solutions is largely determined not only by the system load, but also by the run-time cost of checkpointing and the size of job input and output data. Finally, the throughput and average job execution times generated by CombinedFT for both types of system load are located, as can be observed in the Figures 4.9(a) and 4.10(a), and 4.11(a) and 4.12(a), between respectively the throughputs and average job execution times of FailureDependentRep and MeanFailureCP. This is the logical consequence of the fact that job submissions are

clustered in time and that the heuristic performs some calibrations, after each variation in the system load, before achieving its "optimal" state. Regarding the other observed performance parameters, CombinedFT is almost fully fault-tolerant and results in one of the best average job lengths among the considered algorithms.

4.5 Conclusion

Fault-tolerance forms an important problem in the scope of grid computing environments. To deal with this issue several adaptive heuristics, based on job checkpointing, replication and the combination of both techniques, were designed. The heuristics where evaluated in the DSiDE grid simulator under varying system load and availability. The results have shown that the run-time overhead characteristic to periodic checkpointing can significantly be reduced when the checkpointing frequency is dynamically adapted in function of resource stability and remaining job execution time. Furthermore, adaptive replication-based solutions can provide for even lower cost fault-tolerance in systems with low and variable load, by postponing replication in function of system parameters. Finally, the advantages of both techniques are combined in the hybrid approach that can best be applied when the distributed system properties are not known in advance.

References

[1] D. Feitelson. *Parallel Workloads Archive*. http://www.cs.huji.ac.il/labs/parallel/workload/.

[2] B. Schroeder and G. Gibson. *A Large-Scale Study of Failures in High-Performance-Computing Systems*. In Proceedings of the International Conference on Dependable Systems and Networks (DSN2006), Philadelphia, PA, USA, June 25–28 2006.

[3] S. Hwang and C. Kesselman. *A Flexible Framework for Fault Tolerance in the Grid*. Journal of Grid Computing, 1(3):251 – 272, September 2003.

[4] A. Subbiah and D.M. Blough. *Distributed Diagnosis in Dynamic Fault Environments*. Parallel & Distributed Systems, 15(5):453 – 467, 2004.

[5] Y. Derbal. *A New Fault-Tolerance Framework for Grid Computing*. Multiagent and Grid Systems, 2(2):115 – 133, 2006.

[6] Y. Zhang, M.S. Squillante, A. Sivasubramaniam, and R.K. Sahoo. *Performance Implications of Failures in Large-Scale Cluster Scheduling*. In Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing, pages 233 – 252, New York, USA, 2004.

[7] A.J. Oliner and J. Stearley. *What Supercomputers Say: A Study of Five System Logs*. In Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 575 – 584, Edinburgh, UK, June 25 – 28 2007.

[8] A. Dogan and F. Osgunger. *Matching and Scheduling Algorithms for Minimizing Execution Time and Failure Probability of Applications in Heterogeneous Computing*. Parallel & Distributed Systems, 13(3):308 – 323, 2002.

[9] D.P. Silva, W. Cirne, and F.V. Brasileiro. *Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids*. In Proceedings of Euro-Par, pages 169 – 180, Klagenfurt, Austria, August 26 – 29 2003.

[10] R.Y. De Camargo, A. Goldchleger, F. Kon, and A. Goldman. *Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware*. In Proceedings of the 2nd workshop on Middleware for grid computing, pages 35 – 40, Toronto, Ontario, Canada, 2004.

[11] A.J. Oliner, R.K. Sahoo, J.E. Moreira, and M. Gupta. *Performance Implications of Periodic Checkpointing on Large-Scale Cluster Systems*. In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Denver, Colorado, USA, April 4 – 8 2005.

[12] Y. Li and M. Mascagni. *Improving Performance via Computational Replication on a Large-Scale Computational Grid.* In Proceedings of the 3st International Symposium on Cluster Computing and the Grid (CCGrid), Japan, May 12–15 2003.

[13] C. Bossie and P.M. Fiorini. *On Checkpointing and Heavy-Tails in Unreliable Computing Environments*. SIGMETRICS Performance Evaluation Review, 34(2):13 – 15, 2006.

[14] S. Agarwal, R. Garg, M.S. Gupta, and J.E. Moreira. *Adaptive Incremental Checkpointing for Massively Parallel Systems*. In Proceedings of the 18th annual international conference on Supercomputing, Pittsburgh, PA, November 6 – 12 2004.

[15] S. Chakravorty and L.V. Kale. *A Fault Tolerance Protocol with Fast Fault Recovery*. In IEEE International Parallel and Distributed Processing Symposium (IPDPS), Long Beach, CA, March 26 – 30 2007.

[16] J.W. Young. *A First Order Approximation to the Optimum Checkpoint Interval*. Communications of the ACM, 17(9):530 – 531, September 1974.

[17] E. Gelenbe. *On the Optimum Checkpoint Interval*. Journal of the ACM, 26(2):259–270, April 1979.

[18] A.N. Tantawi and M. Ruschitzka. *Performance Analysis of Checkpointing Strategies*. ACM Transactions on Computer Systems, 2(2):123–144, May 1984.

[19] T. Ozaki, T. Dohi, H. Okamura, and N. Kaio. *Min-Max Checkpoint Placement Under Incomplete Failure Information*. In Proceedings of International Conference on Dependable Systems and Networks (DSN'04), Florence, Italy, June 28 – July 1 2004.

[20] A.J. Oliner, L. Rudolph, and R.K. Sahoo. *Cooperative Checkpointing: a Robust Approach to Large-Scale Systems Reliability*. In Proceedings of the 20th annual international conference on Supercomputing, Cairns, Queensland, Australia, June 28 – July 1 2006.

[21] A.J. Oliner and R.K. Sahoo. *Evaluating Cooperative Checkpointing for Supercomputing Systems*. In Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS'06), Rhodes Island, Greece, April 25 – 29 2006.

[22] Y. Xiang, Z. Li, and H. Chen. *Optimizing Adaptive Checkpointing Schemes for Grid Workflow Systems*. In Fifth International Conference on Grid and

Cooperative Computing (GCC'06), Changsha, Hunan, China, October 21 – 23 2006.

[23] P. Katsaros, L. Angelis, and C. Lazos. *Performance and Effectiveness Trade-off for Checkpointing in Fault-Tolerant Distributed Systems*. Concurrency and Computation: Practice & Experience, 19(1):37 – 63, 2007.

[24] Y. Li and Z. Lan. *Using Adaptive Fault Tolerance to Improve Application Robustness on the TeraGrid*. In Proceedings of TeraGrid'07, Madison, WI, June 4 – 8 2007.

[25] C.J. Hou and K.G. Shin. *Replication and Allocation of Task Modules in Distributed Real-Timesystems*. In Proceedings of Twenty-Fourth International Symposium on Fault-Tolerant Computing, Austin, TX, USA, June 15–17 1994.

[26] S. Choi, M. Baik, J. Gil, C. Park, S. Jung, and C. Hwang. *Group-Based Dynamic Computational Replication Mechanism in Peer-to-Peer Grid Computing*. In Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06), Singapore, May 16 – 19 2006.

[27] A. Ziv and J. Bruck. *Performance Optimization of Checkpointing Schemes with Task Duplication*. IEEE Transactions on Computers, 46(12):1381 – 1386, 1997.

[28] D.K. Pradhan and N.H. Vaidya. *Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture*. IEEE Transactions on Computers, 43(10):1163 – 1174, 1994.

[29] M. Hajdukovic, Z. Suvajdzin, Z. Zivanov, and E. Hodzic. *A Problem of Program Execution Time Measurement*. Novi Sad Journal of Mathematics, 33(1):67–73, 2003.

[30] U. Lublin and D.G Feitelson. *The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs*. Parallel & Distributed Computing, 63(11):1105 – 1122, November 1992. 2003.

5

A dynamic scheduling algorithm for grid workflows

M. Chtepen, F.H.A. Claeys, B. Dhoedt, F. De Turck, P. Demeester and P.A. Vanrolleghem

Submitted to *IEEE Transactions on Parallel and Distributed Systems*.

⋆ ⋆ ⋆

Scheduling of workflows within grid environments is complicated by the heterogeneous and dynamic nature of grid resources, and often incomplete knowledge of workflow parameters. In this chapter we deal with workflows composed of loosely-coupled tasks with input-dependencies for which the execution progress can be monitored at run-time. We introduce an online scheduling algorithm that uses progress information to estimate dynamically the remaining task execution time. Based on the execution time estimates and on the updates on resource load and availability, the algorithm reschedules tasks to minimize the makespan of the workflow as a whole. To evaluate the algorithm's performance, simulation experiments were run with workflow parameters derived from a real world application and system parameters originating from logs of several large scale production parallel environments. The results have shown that the dynamic approach proposed can reduce job makespan by up to 35%.

5.1 Introduction

Grids are composed of heterogeneous, not necessarily dedicated resources that are often managed by different departments or even different organizations. Therefore, characteristic to grids is dynamic and difficult to predict behaviour, whereby resource load and availability strongly vary over time. At the same time, applications or jobs run on grids are often time-consuming, composed of interdependent tasks (workflows). It is clear that under these circumstances the scheduling strategy adopted significantly influences grid performance.

Most of the scheduling approaches currently implemented in operational grid systems are static, which means that the system status is evaluated at particular points in time, and based on this information, the final scheduling decision is taken. Static schedulers are relatively easy to implement but their efficiency is obviously limited by their disability to react to dynamic changes within the execution environment. Therefore, in this work we turn our attention to a more sophisticated dynamic scheduling approach, which makes use of checkpointing and migration mechanisms to reschedule already running tasks when significant system changes occur. However, when designing rescheduling based solutions, we should not forget that migration decisions have to be taken carefully since they can lead to major system overhead. Therefore, a dynamic scheduler has to posses complete and reliable information on system status as well as on workflow execution times. To address the former requirement there exist monitoring tools, such as Ganglia [1], which can provide sufficiently detailed grid status records. Determining workflow execution times, on the other hand, is a much more complicated issue, for which it is difficult to define an effective general solution. The latter can be explained by the fact that when dealing with applications running on grids we have to take into account not only the application complexity and different forms of tasks dependencies, but also the involvement of several grid resources and the dynamics of a grid infrastructure. For this reason, the dynamic scheduling algorithm introduced in this chapter considers only a group of workflow applications for which the execution progress can be monitored at run-time. Based on the progress information, the total workflow execution time on a particular resource is estimated. The proposed algorithm uses these periodic estimates, as well as dynamically collected updates on computational resource status, to reschedule already running tasks. The objective of our method is to minimize the *makespan* of each workflow at the cost of potentially slower execution of individual tasks.

It is important to mention that the algorithm operates on workflows composed of tasks with input-dependencies. Input-dependent tasks, as opposed to MPI-based (Message Passing Interface) tasks [2], do not interact at run-time but instead require inputs generated by other tasks within the workflow before they can be executed. While MPI-based dependent tasks can better be assigned to closely

collocated resources, to reduce the communication overhead, the loosely-coupled input-dependent tasks have more potential for execution optimization by means of intelligent scheduling on widely distributed grid resources, due to the limited number of interactions between tasks.

Finding an optimal schedule within large grid infrastructures, containing hundreds of resources and thousands of jobs running simultaneously, can be extremely time consuming. Therefore, the introduced algorithm is based on a sub-optimal heuristic procedure. Next to the proposed heuristic, we discuss several refinements of the algorithm that all have as a goal the reduction of (re)scheduling overhead.

The algorithm performance is evaluated using the workload model derived from an existing tool for modeling and virtual experimentation with complex environmental systems (Tornado [3]). In particular, we take into account complexity, size of input/output data and variations of execution time predictions of Tornado jobs. The observed values are used to initialize the Lublin job generation model [4] that provides realistic job arrival patterns with a daily cycle, together with the appropriate job length variations.

The remainder of this chapter is organized as follows: in Section 5.2 the related work is discussed; Section 5.3 describes the considered grid and workload models; in Section 5.4 the proposed dynamic algorithm is represented; Section 5.5 discusses some possible algorithm optimizations; in Section 5.6 the performance of the algorithm is evaluated; and, finally, Section 5.7 concludes the article.

5.2 Related Work

Efficient execution of workflow applications in distributed computational environments is recognized by the computer science community as an important and complicated issue. Therefore, in recent years different research efforts were performed that address varying aspects of this problem. The method preferred is in the first place determined by the employed optimization criterion, which can vary from minimization of a workflow application *makespan* [5–13] and optimization of resource usage [14] to guaranteeing a certain quality of service (QoS) [15–17, 17–19] and maximizing operational profit [20].

Another issue is that methods proposed often rely on *a priori* user knowledge or on (semi)automatic prediction mechanisms to determine system and job parameters upon which the scheduling decisions are based. Of particular importance is finding a reliable mechanism for prediction of job execution times, since the latter parameter largely determines the quality of the provided schedule.

Related work on both issues is summarized in the following sections.

5.2.1 Scheduling of Workflow Applications

With regard to scheduling of workflow applications, the available approaches can be subdivided into two categories: static (offline) and dynamic (online).

Static approaches often rely on the so-called ranking concept, where ranks are associated either with tasks or with task-resource pairs. In the first case the assigned ranks determine task scheduling priorities, whereby the task with the highest priority is often submitted to the fastest closest located resource [6–8]. In the second case ranking is applied to sort the available resources in the order of their preference for the execution of a particular task [5, 9]. Within both categories different approaches can be utilized to assign ranks: for instance, *ASKALON* workflow manager [6] ranks tasks based on their average execution time and average communication time between resources of two successive tasks; in [7] a task ranking is computed as the maximum "distance" from this task to the starting node and exiting node in the workflow, where distance refers to the sum of computational and communication costs; in [8] task rank increases with the decrease in difference between the Absolute Earliest Start Time (AEST) and the Absolute Latest Start Time (ALST), which are determined respectively as the maximum of the completion times of the parent tasks and minimum expected start time of the child tasks. ILP-modeling or one of the well-known heuristics from the domain of scheduling parameter sweep applications (Min-min, Max-min and Suffrage) can be applied to construct the full schedule using these individual task rankings.

Using static scheduling within grids has two major disadvantages: (1) *Limited ability to adapt to dynamic changes in grid and task status*. Some of the above mentioned algorithms [7, 8] try to address this issue by dynamically updating task ranking, taking into account resource performance fluctuations and execution time progress of running tasks. Other algorithms refer to advanced resource reservation [9]. However, in both cases the already performed task assignments remain irreversible, which is strongly disadvantageous in situations when performance of some executing resources drops suddenly or new resources are added to the grid; (2) *Accuracy of parameter estimate*. As was mentioned earlier, performance of static approaches mainly depends on the quality of job execution time estimates and resource performance prediction. Unfortunately, values of both parameters are extremely difficult to determine in advance, which often leads to poor algorithm performances.

As opposed to static approaches, dynamic algorithms rely on rescheduling to improve task-resource mapping when dynamic system changes occur. In [10, 12] dynamic approaches are discussed that perform the initial task assignment based on resource performance and job execution models, constructed using historical information on previous application runs. Afterwards, the algorithms keep track of dynamic system updates to perform rescheduling and to modify the historical information repository to improve the estimate accuracy in the subsequent planning.

The major difference between the algorithms lies in fact that rescheduling decisions are taken using different sets of performance parameters: in [10] resource load is monitored; while in [12] batch queue wait time of each resource, network bandwidth and disk write speed are considered. Furthermore, in [10] rescheduling is triggered each time new parameters can lead to reduction of workflow *makespan*, while [12] tries to reduce the rescheduling overhead by omitting migrations when performance degradation due to dynamic changes does not exceed a particular tolerance threshold. In [21] three dynamic application level scheduling algorithms for workflows are proposed: Application Modified Critical Path (AMCP), Application Improved Critical Path using Descendant Prediction (AICPDP) and Application Highest Level First with Estimated Times (AHLFET). The algorithms are modified versions of the three often used low level DAG scheduling methods: Modified Critical Path (MCP), Improved Critical Path using Descendant Prediction (ICPDP) Algorithm and Highest Level First with Estimated Times (HLFET) Algorithm. While the original methods are static, the modified algorithms take their scheduling desisions dynamically, based on execution statistics of previous tasks in the present workflow. Yet another dynamic workflow assignement approach for grids, relying on graph partitioning, is discussed in [22]. The approach considers QoS-based scheduling with focus on communication overhead and trust relationships between grid nodes to provide a trust, cost and deadline based schedule (TCD).

5.2.2 Execution Time Estimation

Most of the above mentioned studies assume that exact task execution times are somehow known in advance or can be provided by the end-user [7–9]. Others make use of historical information [10] or apply *application component performance modeling* [5], which implies observation of the number of floating point operations executed by the application as well as its memory access pattern. However, both latter approaches are complex and error prone. The historical approach requires a large number of historical records to be stored before matchmaking of an executed application against the available entries provides a sufficiently accurate estimate. However, the more records are stored in the database, the longer the matchmaking procedure takes. On the other hand, estimate of the number of floating point operations and the memory access pattern is neither trivial, since the course of the applications execution can vary dynamically, and is, furthermore, strongly dependent on input parameters.

Several studies were explicitly dedicated to application execution time estimate. In [23] the ScoPred performance predictor is discussed. ScoPred applies multiple linear regression using rough estimates of application execution time provided by the end-user and historical application-run data to predict the execution

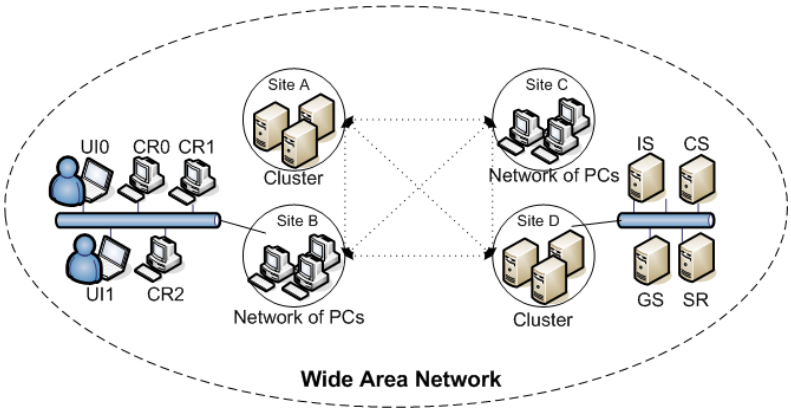


Figure 5.1: Considered grid model: Computational Resource (CR), Grid Scheduler (GS), User Interface (UI), Information Service (IS), Checkpoint Server (CS), Storage Resource (SR).

times. A similar approach is proposed in [24], where an online workflow application execution time prediction system relies on historical records, classified using *similarity templates*. A template refers to a set of selected workflow attributes. Matchmaking of templates is supervised by an *expert user*, who is supposed to indicate relevance of each attribute for a particular application.

5.3 Grid and Workload Models

The design of efficient, autonomic and widely applicable mechanisms for workflow applications running on grids is an extremely difficult issue. Therefore, in this article we make several assumptions, with respect to the scheduled applications and grid architectures. The properties of the considered workflow and the distributed environment are discussed in details in the following subsections.

5.3.1 Grid Model

The grid model considered in this work (see Figure 5.1) consists of geographically dispersed heterogeneous Sites (S), containing an arbitrary number of Computational Resources (CRs). CRs can either be dedicated machines or simply personal computers, used to run grid applications during idle periods. Consequently, CRs have different computational capacities and load on the resources can strongly vary over time. Since the actual resource capacity is determined by various parameters, for simplicity, we use an often applied theoretical metric, called MIPS, to compare the performance of resources in our model.

Another simplifying assumption is taken regarding CR sharing among different grid tasks, where we assume that each task submitted to a resource receives a fair share of its computational capacity. However, besides grid jobs, background load $MIPS^{ex}$ is imposed on some CRs. Therefore, the speed $MIPS_{CR}$, with which the computational resources CR can run some grid task is determined by the following equation:

$$MIPS_{CR} = \frac{MIPS_{CR}^{total} - MIPS_{CR}^{ex}}{n_{CR}}, \tag{5.1}$$

where $MIPS_{CR}^{total}$ stand for the total resource capacity and n_{CR} is the number of grid tasks running on CR.

To avoid resource overload and consequently the stagnation of the progress of all tasks running on it, we define the maximum number of grid jobs that can be run simultaneously on a resource n_{CR}^{max}. When this maximum number is reached, tasks submitted to the resource are rejected.

Next to CRs, the assumed grid infrastructure includes a number of general services, such as a Grid Scheduler (GS), which maintains a job queue and is responsible for task-resource matchmaking; several distributed User Interfaces (UIs), utilized by end-users to submit their applications to the grid; an Information Service (IS) required to collect information on grid status; a Checkpoint Server (CS) where checkpoints are saved; and a Storage Resource (SR) where output data is transferred after job execution. Important to mention is that an elaborate grid infrastructure usually includes several SRs, which requires an implementation of an appropriate SR selection algorithm. The latter is, however, not the focus of the present article and therefore we limit the considered grid architecture to a single SR.

We take into account most of the operational overhead of the above-mentioned components and interactions between them to provide for a realistic environment for execution of workflow applications. The modeled overheads are based on the overhead analysis for scientific workflows presented in [25]. According to this study, the overheads can be subdivided into the following four categories:

- **Middleware overhead** relates to the overhead of grid middleware services, such as the time to query the IS, the time to predict task execution period, the time to compute a schedule (scheduling overhead), the time to recalculate the previous schedules (rescheduling overhead), the time to prepare tasks for distribution on a grid and the time to checkpoint / rollback workflows. These overheads are included in our model in the form of constant delays of typically several seconds.
- **Data transfer** overhead mainly originates from input / output file staging and additional transfers upon workflow rescheduling and rollback. In our grid model all sites are interconnected by a Wide Area Network (WAN),

while the communication within the sites go by different Local Area Networks (LANs). Network overhead of transferring a task J is calculated as stated in the formula (5.2), where r is the network route (or a sequence of links) between the old and the new resources; In_J and $CSize_J$ are respectively input data size and checkpoint size of the task J; L_l is latency of a link l; and $\lambda_r^{l_b}$ is the bandwidth allocated to route *r* when $l_b \in r$ is the bottleneck link, *i.e.* the link determining the transmission rate of the data.

$$NO_r^J = \frac{In_J + CSize_J}{\lambda_r^{l_b}} + \sum_{l \in r} L_l \tag{5.2}$$

$$= \frac{(In_J + CSize_J) \times \sum_{l_b \in \tilde{r}} \frac{1}{\phi_{\tilde{r}}}}{\frac{1}{\phi_r} \times B_{l_b}} + \sum_{l \in r} L_l, \tag{5.3}$$

It is assumed that network transfers within LANs originate exclusively from grid jobs, while WANs are open to background network traffic. This difference leads to varying models for bandwidth (B) allocation within network links. We use the network model described in [26] and assume that every data transfer route going through a WAN link gets a small equal share of the total link capacity, while capacities of LAN links are proportionally shared among simultaneous active grid transfers according to the INV-RTT-BOUNDED model, which allocates to each flow r a share of the bandwidth of the bottleneck link l_b with a weight $\omega = 1/\phi_r$, where ϕ_r is the round-trip time of route r that is calculated as $\phi_r = 0.2 \times \sum_{l \in r} L_l$. Applying the INV-RTT-BOUNDED model we can transform the formula (5.2) to the formula (5.3). The latter states that $\lambda_r^{l_b}$ depends on the capacity of l_b (B_{l_b}) and on the round-trip time ϕ of each route going through l_b. In contrast to the often applied fair bandwidth sharing model, where bandwidth is equally shared among all active data transfers, we compute $\lambda_r^{l_b}$ in such a way that shorter transfers get more bandwidth assigned, which is a better approximation for the TCP-protocol [26]. For simplicity we assume that total link capacities do not change over time and that links are not subject to failure.

- *Loss of parallelism* happens, for instance, when a grid does not contain sufficient resources to run all parallel tasks simultaneously (*serialization*), or when some parallel computational activities finish faster than others, leaving the allocated processors idle (*load imbalance*). In the case of input-dependent workflows, we can talk about *load imbalance* when there is a difference between the execution times of several in parallel running tasks, inputs from which are required for processing of another task(s) within the workflow. This issue is explained in detail in the following sections.
- *Activity overhead* results from background load imposed on grid resources, or from the loss of computations due to re-execution of computational activities when failure occurs, or when a new powerful site is added to the

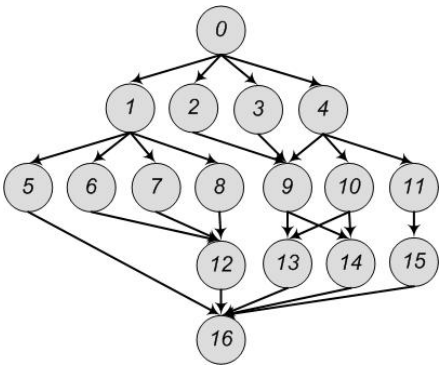


Figure 5.2: Example of a workflow with input-dependencies, organized into a DAG structure.

system. In this work we do not consider resource failure and dynamic site arrival explicitly. However, similar effect is achieved by allowing strong variations in resource load.

5.3.2 Workload Model

Workflows considered in this work can be represented as a Directed Acyclic Graph (DAG) $\mathcal{W} = (Nodes, Edges)$ with a single *initial* or *root* node and a single *final* node. Figure 5.2 shows an example of a possible dependency structure, where $Nodes = \cup_{i=0}^{n} J_i$ indicate application tasks and $Edges = \cup_{0 \leq j < k \leq n}(J_j, J_k)$ show the dependency flows. In concreto, the task from which an edge departs (*parent task*) generates output data that serves as input into the task where the edge arrives (*child* or *dependent task*). In this work we only consider input-dependencies between tasks, since this type of dependencies imply relatively limited communication overhead and allow for assigning interdependent tasks to resources widely distributed within a global grid environment. Furthermore, it is assumed that jobs submitted are batch jobs with the dependency hierarchy fixed *a priori*.

Important to mention is that the considered dependency structure often occurs in real world applications. For example, complex simulations with varying parameter values can be represented as a single *initial* task generating inputs (parameter values) for an arbitrary number of *dependents*, outputs of which can in turn be either grouped together or split to serve as inputs for *dependent* tasks on the next level of the dependency graph. There can be an arbitrary number of dependency levels but the temporary outputs on intermediate levels should be combined into a single *final* task, where the definite application result is calculated.

In general, when a task is submitted into the grid, the system either does not posses any information on its execution time, or, at best, there is a rough estimate

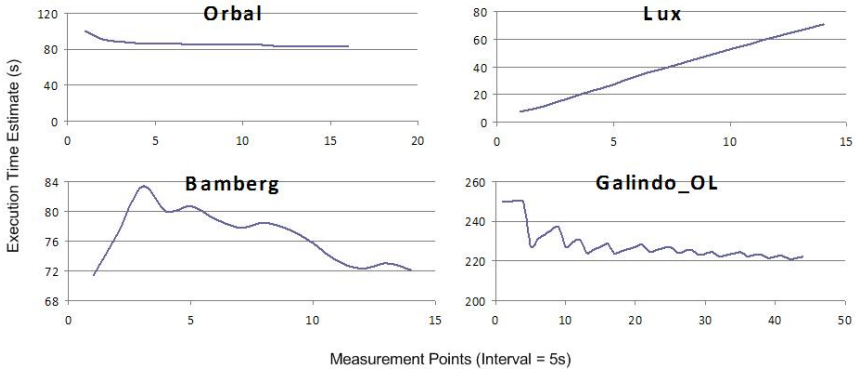


Figure 5.3: Examples of evolution of execution time estimates for the "Orbal", "Lux", "Bamberg" and "Galindo_OL" simulation experiments.

of the task complexity available. However, there exists a relatively broad group of applications for which the execution progress can be monitored at run-time. In this article we consider this type of workloads and we use dynamically collected information on task progress and resource parameters to periodically estimate task execution times. The reference (*i.e.* resource independent) execution time of task J can be estimated as

$$E_J^{est} = \frac{100\% \times T_J \times (MIPS_{CR}^{total} - MIPS_{CR}^{ex})}{P_J \times n_{CR}}, \tag{5.4}$$

where we assume that J is currently running on some computational resource CR, P_J is the percentage of the task workload already completed and T_J is the execution time thus far. E_J^{est} can be interpreted as the estimated execution time on a theoretical reference machine, which is assumed to have $MIPS = 1$ and $n = 1$. Periodically constructed estimates can strongly vary over time, but in general, the longer the task is running the better is the estimate.

To construct a realistic model describing the evolution of the execution time estimates, the behaviour of a large number of experiments originating from the Tornado tool was observed. From these observations it can be seen that the courses of the prediction curves show strong variations as experiment type, input parameters and used solvers change. Figure 6.2 shows the estimate evolution for four Tornado simulations, respectively called "Orbal", "Lux", "Bamberg" and "Galindo_OL" [27]. However, all the observed curves can be categorized according to following approximation models:

- **Overestimate** means gradual decrease of the estimated values until the *stable state* is reached. In the *stable state* the exact execution time is known

and the prediction does not change any longer."Orbal" is a simulation experiment following the *overestimate* model.

- **Underestimate** refers to gradually increasing estimates. "Lux" is a good example of this model.
- **Fluctuating** stands for predictions that are oscillating over time or exhibit rather an erratic pattern. In the cases of "Galindo_OL" and "Bamberg" we can talk about the *fluctuating* model.

In this article we generalize the above described theoretical models using the following mathematical descriptions: $E_J^{est} = E_J^{ref} + r_1 A e^{-bt}$ describes an exponentially decreasing *overestimate* model; $E_J^{est} = E_J^{ref} - r_1 A e^{-bt}$ stands for an exponentially increasing *underestimate* model; and $E_J^{est} = E_J^{ref} \pm r_1 A e^{-bt} sin(2\pi Ft + r_2\varphi)$ represents *fluctuating* evolution of execution time estimates. In the above equations E_J^{ref} stands for the reference total execution time of the task J; r_1 and r_2 are random weight factors that can take values between 0 and 1; A is the amplitude of the estimates' oscillations; t is the current simulated time; b is a weight factor that determines the speed in the decrease / increase of A over time; F stands for the oscillation curve period; and, finally, φ represents the oscillation curve phase.

5.4 Dynamic Scheduling Algorithm

The overhead analysis of workflow applications presented in [25] were verified using the WIEN2k package for performing electronic structure calculations of solids [28]. The results suggested that the largest overhead originates from *serialization*, when a grid has a frequent deficit of free resources to run parallel jobs simultaneously. *Serialization* overhead can be extremely high (up to 90%) when the system is badly dimensioned and approaches zero when the grid includes a sufficient number of resources. If extending the grid size is not an issue, this type of overhead can be neglected.

The second largest overhead results from *load imbalance*, followed by *data transfer* overhead.

Obviously, the exact impact of both types of overhead strongly depends on properties of the considered application: variation in computational complexity of parallel tasks, size of inputs / outputs, size of checkpoints, *etc*. However, the study certainly serves as an indication of the importance of different overheads.

The algorithm proposed in this chapter mainly addresses the *load imbalance* overhead issue. It reschedules parallel tasks at run-time, based on dynamically collected updates on system status and task progress. While making (re)scheduling decisions our method also takes into account other categories of overhead, such as

data transfer, migration and restart overheads. The objective is to reduce the overall application *makespan* by finding a good balance between rescheduling overhead and more efficient utilization of the available resources.

In the following sections the dynamic scheduling approach is introduced in several phases. Initially the basic version of the algorithm is discussed, followed by several important optimizations.

5.4.1 Basic Dynamic Scheduling Algorithm

Basically, in each scheduling iteration all idle/failed *root* tasks as well as idle/failed *dependent* tasks, whose *parents* have finished their execution, can be considered for scheduling on the available CRs. A set of such tasks at each point in time is called a *ready* or a *parallel set*. If we assume that we operate on an ordered set of nodes, a *parallel set* can be formally defined as

$$RS(t) = \{J_i \in Nodes \mid J_i = Idle \wedge (\neg\exists(J_k, J_i) \in Edges \vee$$

$$\forall(J_k, J_i) \in Edges : J_k = Done(t)) \wedge k < i\}.$$

However, to speed up the execution of workflows, we have to take into account dependency structures. Therefore, we introduce the notion of a *parent set*. A *parent set* is a set of tasks generating input for the same group of *dependents*:

$$PS_{J_i} = \{J_k \in Nodes \mid (k < i) \wedge (J_k \in RS(t))$$

$$\wedge(\exists(J_k, J_i) \in Edges)\}.$$

For example, in Figure 5.2 tasks $\{0\}$, $\{1\}$, $\{2,3,4\}$, $\{4\}$, $\{6,7,8\}$, $\{9,10\}$, $\{11\}$, $\{5,12,13,14,15\}$ form *parent sets* for respectively tasks 1-4, 5-8, 9, 10-11, 12, 13-14, 15 and 16. Clearly, each task in the considered workflows, except for the *initial* task, has a *parent set*. *Parent sets* of different tasks are not necessarily unique and they can overlap, which is the case for the sets $\{2,3,4\}$ and $\{4\}$ in the example above.

The idea behind the proposed algorithm is to minimize the execution time of a *parent set* as a whole, but at the same time to guarantee that the included tasks finish more or less simultaneously.

In real-world applications, individual tasks within workflows often have significantly varying execution times, which can result in imbalanced execution of a *parent set*, whereby some tasks finish much faster than others. However, their *dependents* can not proceed with the execution, since they require inputs from the whole *parent set*. Therefore, it seems desirable to schedule short tasks on slower resources, as long as it does not delay the execution of the *parent set*, and leave

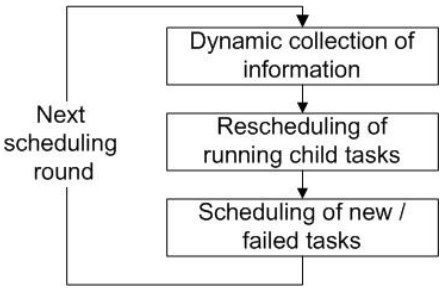


Figure 5.4: Flow of the operation phases of the proposed dynamic scheduling algorithm.

faster CRs to tasks requiring fast processing. Our optimization heuristics can be described by the following equations:

$$\min_{\forall PS \in \mathcal{W}} (\max_{\forall J_i \in PS} \{E^{est}_{J_i}\}) \tag{5.5}$$

$$\forall PS \in \mathcal{W} : \min_{\forall J_i, J_k \in PS} \mid E^{est}_{J_i} - E^{est}_{J_k} \mid \tag{5.6}$$

, where E_{J_i} and E_{J_k} stands for execution times of tasks J_i and J_k respectively.

In concreto, the operation of the proposed algorithm can be subdivided into three iterative steps (see Figure 5.4): collection of dynamic information, rescheduling of running PSs, scheduling of *ready* tasks.

5.4.1.1 Step 1: Collection of Dynamic Information

At the beginning of each scheduling round, the algorithm requests a grid status update from the IS. In particular, the information on speed and resource availability (failure, restart, new CR registrations), as well as the information on job status and progress of running jobs is collected. Here we have to take into account that the collected data can sometimes be outdated, leading to "erroneous" scheduling decisions. The severity of these faults will mainly depend on the dynamics of the grid at hand, the length of the interval used by the IS to query the grid, and the data transfer delays.

5.4.1.2 Step 2: Rescheduling of Running PSs

In this section we gradually explain the functionality of the rescheduling step of the algorithm, using pseudocode. In this step, *running PSs* (RPSs) are reassigned to balance the execution times of the included tasks, based on newly collected updates on tasks progress and grid status. Under the term *running PS* we understand a PS that exclusively contains finished tasks and tasks actually running on active grid resources. Clearly, to be rescheduleable an RPS should contain at least one non-finished task.

The pseudo code below describes the operation of the proposed algorithm (see Table 5.1 for a listing of symbols):

Input: Set of current RPS's:$U = \{\rho_1, ..., \rho_n\}$
Output: Set of current *available* CR's:$R = \{\tau_1, ..., \tau_m\}$
repeat
  $\rho_{shortest} \Leftarrow getSmallestSet(U)$
  $\xi = \emptyset, \chi = \emptyset, \xi_{max} = 0$
  for all $\eta_i \in \rho_{shortest}$ **do**
    $\tau_{cur} \Leftarrow getCurrentCR(\eta_i)$
    $\xi_i \Leftarrow RE(\eta_i, \tau_{cur})$
    $\xi \Leftarrow \xi \cup (\xi_i, \eta_i)$
    if $\xi_i > \xi_{max}$ **then**
      $\xi_{max} \Leftarrow (\xi_i, \eta_i)$
    end if
  end for
  repeat
    $\tau_{cur} \Leftarrow getCurrentCR(\eta_i)$
    $\tau_{best} = \emptyset, \xi_{best} = 0$
    if $\xi_i \equiv \xi_{max}$ **then**
      $\xi_{min} = \xi_{max}$
      for all $\tau_j \in R$ **do**
        $\xi_{temp} \Leftarrow RE(\eta_i, \tau_j) + NO(\eta_i, \tau_{cur}, \tau_j)$
        if $\xi_{temp} < \xi_{min}$ **then**
          $\tau_{best} \Leftarrow \tau_j, \xi_{min} \Leftarrow \xi_{temp}$
        end if
      end for
      $\xi_{best} \Leftarrow \xi_{min}$
    else
      $\sigma_{min} = \xi_{max} - \xi_i, \xi_{new} = 0$
      for all $\tau_j \in R$ **do**
        $\xi_{temp} \Leftarrow RE(\eta_i, \tau_j) + NO(\eta_i, \tau_{cur}, \tau_j)$
        if $\xi_{max} - \xi_{temp} < \sigma_{min}$ **then**
          $\tau_{best} \Leftarrow \tau_j, \sigma_{min} \Leftarrow \xi_{max} - \xi_{temp}$
          $\xi_{new} \Leftarrow \xi_{temp}$
        end if
      end for
      $\xi_{best} \Leftarrow \xi_{new}$
    end if
    if $\tau_{best} \neq \emptyset$ and $\tau_{cur} \neq \tau_{best}$ **then**
      $D \Leftarrow getDependentTasks(\tau_{best})$
      if $\neg delayExecution(D)$ **then**

Parameter	Value
U	Set of currently available RPS's
R	Set of currently available CR's
$\rho_{shortest}$	RPS with smallest number of running tasks
η_i	i^{th} task from $\rho_{shortest}$
ξ	List of remaining execution times of tasks within $\rho_{shortest}$
ξ_i	Remaining execution time of task i
ξ_{max}	Maximum remaining execution time within ξ
ξ_{best}	"Optimal" execution time of a task
ξ_{min}	"Minimum" execution time of a task
τ_{cur}	CR executing η_i
τ_{best}	Best CR to execute a particular task
σ_{min}	Minimum difference with execution time of the longest task
D	Set of *dependent* tasks
C	Set of *independent* tasks
χ	List of tasks already assigned within $\rho_{shortest}$

Table 5.1: Listing of symbols

40: $C \Leftarrow getIndependentTasks(\tau_{best})$
41: $checkpoint\&Cancel(C, \tau_{best})$
42: $submitToFastestAvailableResources(C, R)$
43: $checkpoint\&Cancel(\eta_i, \tau_{cur}), submit(\eta_i, \tau_{best})$
44: $update(R), \xi \Leftarrow \xi \cap \neg(\xi_i, \eta_i) \cup (\xi_{best}, \eta_i)$
45: **else**
46: $R \Leftarrow R \cap \neg\tau_{best}$
47: **end if**
48: **else**
49: $\chi \cup (\xi_i, \eta_i)$
50: **end if**
51: **until** $\chi \equiv \xi$ or $R \equiv \emptyset$
52: $U \Leftarrow U \cap \neg\rho_{shortest}$
53: **until** $U \equiv \emptyset$

A set of RPSs containing non-finished tasks, together with a set of the *available* resources form the input for the rescheduling algorithm (Lines 1 – 2). However, it is important to mention that we omit RPSs containing a single *root* task. The point is that *root* tasks are independent of any other tasks and thus can be scheduled immediately after their arrival into the GS queue. If the submission frequency is high, *root* tasks will occupy to a large extent fast grid resources, delaying the

execution of *dependent* tasks.

To find a solution for the problem stated in Formulas 5.5 and 5.6 we can match tasks within each *running PS* against each *available* resource. *Available* refers in this context to a non-failed resource with $n_{CR} \leq n_{CR}^{max}$. This approach basically leads to solving a *m*-combinatorial problem, where we select *m* resources (to run *m* tasks) from a total set of *n* resources. The computational complexity of solving this *m*-combinatorial problem can be approximated as *O(n!)*, which means that even relatively small values of *n* lead to a combinatorial explosion. To reduce the search space, we propose a heuristic search method that instead of processing all *running PSs* at one time, assigns the latter to the available resources in the order of the increasing number of tasks within a set (Lines 4 – 13). Assigning a higher priority to small *parent sets* advances job throughput, since short or almost finished jobs are scheduled to the fastest available resources. In concreto, the algorithm iterates over the set *U*, selecting in each iteration the RPS with the smallest number of running tasks (function *getSmallestSet(U)* in the pseudocode above). For each task J within the shortest PS the algorithm applies the formula (5.4) on dynamically collected task progress information to find the expected total execution time E_J^{est}. Knowing the latter and the execution progress P_J, we can easily calculate the remaining execution time $RE_{J,CR}^{est}$ of J on each available resource CR:

$$RE_{J,CR}^{est} = \frac{(E_J^{est} - E_J^{est} \times P_J/100\%) \times n_{CR}}{MIPS_{CR}^{total} - MIPS_{CR}^{ex}}. \tag{5.7}$$

In the second iteration over the shortest PS, the algorithm tries to balance the execution times of already running tasks, taking into account updates on resource status and task progress. First of all, the algorithm searches for the fastest currently *available* resource to execute the task with the longest remaining execution time (Lines 14 – 26). Secondly, for each task with shorter remaining execution time, the algorithm tries to find a resource that minimizes the difference (σ_{min}) with the execution time of the longest task (Lines 27 – 37).

It is important to mention that not only *available* resources are considered for task rescheduling but also non-failed resources running *root* tasks. Since the execution of a *dependent* task has higher priority than the execution of a *root* tasks, the latter can be interrupted if it is running on the resource that is the best suitable for processing of the *dependent* task. Therefore, the described rescheduling algorithm considers the capacity of each *available* resource with and without *root* tasks running on it. Besides, when considering a task migration we have to take into account not only the computational speed up that can be achieved on a faster or less loaded resource but also the network transfer overhead NO_r^J.

Finally, if a better resource for the execution of some *dependent* task is found, the algorithm checks if the rescheduling does not delay the execution of other *dependent* tasks eventually running on this resource (function *delayExecution(D)*

in the pseudocode above). If a migration of the task causes a slow down in the execution of some RPS, the migration is canceled and the algorithm looks for a new candidate resource. On the other hand, when there is no delay for other RPSs the algorithm can proceed with the task migration. At that time some of the *root* tasks running on the selected resource can in their turn be interrupted and migrated to the fastest still available resource ($checkpoint\&Cancel(C, \tau_{best})$. If, however, the *root* tasks can not be reassigned to another resource without delaying the execution of some RPS, then their execution is postponed (Lines 38 – 51).

5.4.1.3 Step 3: Scheduling of Idle or Failed Tasks

After the rescheduling phase, the algorithm proceeds with the assignment of *ready* tasks to the remaining resources. The tasks are processed in the order of their arrival into the GS-queue. If sufficient CRs are available, the algorithm tries to schedule the whole *parent set*, where the next to schedule task belongs, in a single iteration. When a *parent set* can only partially be processed, it gets the highest priority in the next scheduling iteration.

For some applications and particularly for failed tasks we may posses an initial estimate of the total tasks execution time. In this case the scheduler proceeds as described in Step 2. Otherwise, tasks are assigned randomly.

5.5 Dynamic Scheduling Algorithm Optimizations

The dynamic scheduling approach introduced in the previous section has a complexity of $O(k^2n)$, where k is the number of RPSs to be scheduled and n is the number of grid resources. This complexity is significantly lower than $O(n!)$, imposed by the optimal solution. However, for large and heavily loaded grids an online matchmaking of each task against each *available* resource remains a time consuming operation. Furthermore, task migrations have a considerable impact on performance of most applications, as they require the application task execution to be interrupted, checkpointing to be performed, data (input and checkpoint) to be transferred over the network to a new resource and the task to be restarted. If not applied carefully, rescheduling overhead can exceed the benefits of adaptation to dynamic system changes.

In this section we discuss several possible optimizations that have as a goal the improvement of the algorithm performance into two directions: (1) search space reduction and (2) elimination of *unprofitable* task migrations.

5.5.1 Total Remaining Execution Time Monitoring

Since migration costs time, it can be omitted in case a task approaches the end of its execution.

Currently, the algorithm considers task rescheduling when the remaining execution time on the current resource exceeds the sum of the execution time on another resource and data transfer time between both resources. However, we do not consider the task checkpointing overhead (*C*) and the restart time (*R*). Values of both parameters can vary from several milliseconds to several minutes, depending on the checkpointing mechanism utilized and the complexity of the application at hand. Besides the fact that migration slightly enlarges task execution time, it can also indirectly delay the execution of other scheduled tasks by consuming bandwidth and occupying fast resources, but also by initiating rescheduling of running *root* tasks.

Therefore, we expand the algorithm with an additional control statement $\xi_{cur} > RE^{min}$ before proceeding with the rescheduling step (see Section 5.4.1.2). Here ξ_{cur} is the expected execution time on the CR where the task η_i is currently running and RE^{min} is the minimum remaining execution time required to proceed with the rescheduling.

5.5.2 Migration Profit Prediction

This approach performs task migration only if the currently predicted execution time is expected to be improved with at least a certain predefined percentage γ. Therefore, for the longest task within a PS the rescheduling criterion becomes $\xi_{best} + \gamma \times \xi_{cur} < \xi_{cur}$, while the migration of other tasks to some resource CR is considered only if $(\xi_{max} - \xi_{CR}) + \gamma \times \sigma_{min} < \sigma_{min}$.

5.5.3 Oscillation Pattern Detection

Is Section 5.3.2 several possible execution time prediction models are discussed. One of them is a frequently occurring *fluctuating* model. The issue with this model is that strong periodic variations in execution time prediction can lead to frequent rescheduling of PSs. It can be advantageous for these tasks to keep track of their average estimated execution time E^{avg} and use it instead of E^{est} to filter out oscillations, providing for smoother flow of the prediction curves and reducing unnecessary rescheduling.

5.5.4 Resource Grouping

Instead of consecutively matching each task against each *available CR*, we group resources according to similarity of their properties. Therefore, the original algorithm is modified as follows:

- All *available* resources are initially subdivided into a number of groups ($G_{\#}$), with a maximum of $\theta = \frac{n}{G_{\#}}$ resources within each group. Since the scheduling decisions taken by the proposed algorithm are mainly based

on resource speed and location, we group CRs based on their MIPS characteristic and LAN assignments: $G_{\#}$ most *collocated* resources with similar $MIPS_{CR}$ are assigned to the same group. $G_{\#}$ can either be specified by the end-user or derived automatically, based on size and topology of each grid system at hand, as well as on variations of resource capacities. In general, the more groups we have, the larger is the search space, but the more accurate is the provided schedule.

Important to mention is that CRs within each group are sorted in the order of increasing capacity, while groups themselves are sorted in the order of the increasing capacity of the included resources.

- For the longest task η_{max} within each considered PS (see Section 5.4.1.2) we compute its minimum remaining execution time ξ_{min} as

$$\xi_{min} = \min_{\forall g \in G}(RE(\eta_{max}, \tau^g_{max}) + NO(\eta_{max}, \tau_{cur}, \tau^g_{max})),$$

where *G* is a list of all resource groups and τ^g_{max} is the "fastest" resource within each group.

- For the other tasks η_i (see Section 5.4.1.2) we calculate ξ_{temp} for each group $g \in G$ as

$$\xi_{temp} = RE(\eta_i, \tau^g_{avg}) + NO(\eta_i, \tau_{cur}, \tau^g_{avg})),$$

where τ^g_{avg} is the "average" resource within the group g. The speed of the "average" resource is calculated as

$$MIPS_{\tau^g_{avg}} = \frac{\sum_{i \in g} MIPS_i}{\theta}.$$

Based on the average resource speed within each group, the algorithm can decide to migrate a task to a new group of resources. In this case, the task is scheduled on a random CR within the new group, where it does not delay the execution of other PSs.

- After each (re)scheduling procedure, modified resources are reinserted into the appropriate positions into the *G*-list.

Due to resource grouping the algorithm complexity is reduced from $O(k^2n)$ to $O(nlog(n) + k^2G_{\#})$.

5.6 Dynamic Algorithm Evaluation

In this section we evaluate the performance of the proposed dynamic scheduling method in its original form and when it is optimized as discussed in the previous

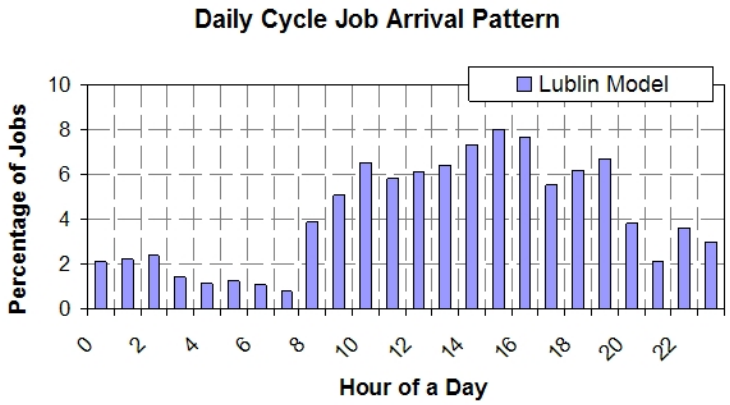


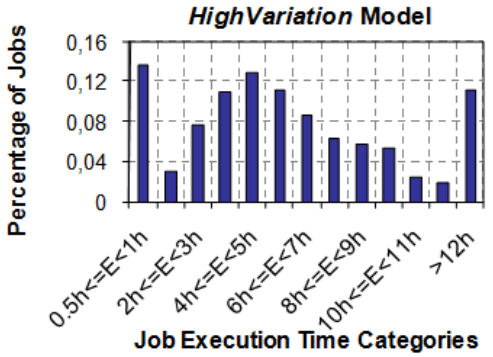
Figure 5.5: Job arrival pattern with daily cycle.

section. The evaluations are performed in the DSiDE [29] grid simulation environment. DSiDE was chosen among other existing grid simulation tools (GridSim [30], SimGrid [31], NSGrid [32]) because of its high flexibility with respect to modeling and simulation of dynamic grid events, such as resource failure, resource and network load variations, changes in job execution progress, *etc*.

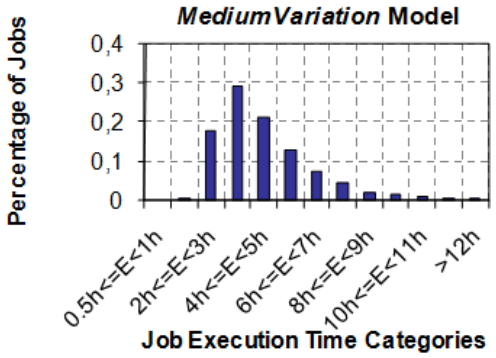
5.6.1 Experiment Description

Our simulation scenarios operate on a grid system including 4 widely distributed sites, each containing 32 CRs that can execute a maximum of 2 grid tasks simultaneously (see Table 5.2 for a listing of parameter values). Tasks are run with speeds varying between 0.1 and 4 MIPS. The network model discussed in Section 5.3.1 is applied: within WAN links each data transfer is assigned an equal share of 5 Mbit/s of the available bandwidth, while within LAN links 1 Gbit/s transfer capacity is divided among the active transfers within the link. Additionally, WAN link latency varies between 3 and 10 ms, and LAN links possess fixed 1 ms data transfer latency. Next to data transfers, running intervals of the GS and the IS are important sources of system delays. In our case, rescheduling and new task assignment procedures are run every 10 min, while grid status information in the IS is updated every 5 min.

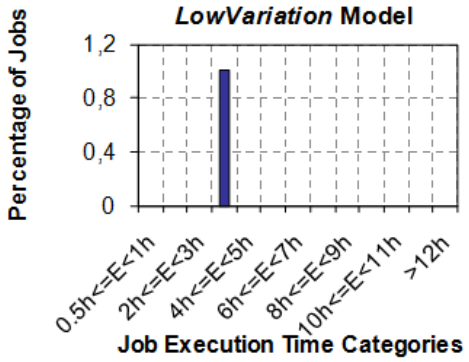
It is difficult to determine a general and realistic approach to describe load variation of grid resources, since it depends on the nature of a resource at hand (dedicated machines, PCs, *etc*.), the type of applications run and user behaviour patterns. Therefore, we simply assume that load variations occur periodically every 20 min and that the load can increase or decrease with a certain percentage P



(a)



(b)



(c)

Figure 5.6: Task execution time variation models: (a) high execution time variation between dependent tasks; (b) medium execution time variation between dependent tasks; (c) low execution time variation between dependent tasks.

Parameter	Value
A	U($E_{ref} \times 150\%$, $E_{ref} \times 200\%$)
b	U(0, 1)
B_l, (for LAN links)	1 GBit/s l
C	2 s
$CSize_J$	1 GB
F	U($E_{ref} \times 5\%$, $E_{ref} \times 10\%$)
I_J	1 GB
$Interval_{CRLoadVar}$	20 min
$Interval_{GSched}$	10 min
$Interval_{IS}$	5 min
$\lambda_r^{l_b}$ (for WAN links)	5 MBit/s
L_l (for WAN links)	U(3 ms, 10 ms) ms l
L_l (for LAN links)	1 ms l
$MIPS_{CR}^{total}$	U(0.1 MIPS, 4 MIPS)
n	128
n_{CR}^{max}	2
O_J	1 GB
φ	U($E_{ref} \times 1\%$, $E_{ref} \times 2\%$)

Table 5.2: Listing of parameter values

of the original load. *P* is modeled to be uniformly distributed between 1 and 100%.

Arrival of applications into the considered grid environment follows the Lublin submission model. This model describes a daily cycle job arrival pattern, often observed in production parallel environments [4]. According to the Lublin model, most of the jobs (about 80%) arrive during the day time, between 8 AM and 8 PM, as shown in Figure 6.6.

As was mentioned previously, the characteristics of modeled jobs are derived from an existing tool for modeling and virtual experimentation with environmental systems, called Tornado. Tornado possesses a broad category of virtual experiments with input-dependent tasks, whereby all Tornado input-dependencies can be subdivided into two groups:

- **Parameter sweep**: sub-experiments (Tornado equivalent for a task) are run on the same model but with different parameters, which results in similar execution times between the tasks.
- **Model sweep**: sub-experiments are run on different models, often leading to strongly varying execution times.

Therefore, in our simulation experiments we consider the following possibilities for execution time variations between *parallel* tasks: tasks have strongly varying execution times (*HighVariation* model); tasks execution time variations are moderate (*MediumVariation* model); task execution times are practically similar (*LowVariation* model). The parameters of the three models were chosen such that the execution times generated correspond to the observed execution times of Tornado experiments executed on UGent-grid [33]. UGent-grid is a part of the Belgian grid infrastructure and it consists of 76 CRs having a total of 222 CPUs, 304 GB memory and 4.4 TB disk space. For outcome of the three models, refer to Figure 5.6, where the execution times on the graphs are shown for the case when all tasks are executed on some reference machine CR^{ref} with $MIPS_{CR^{ref}} = 1$ and $n_{CR^{ref}} = 1$.

Finally, we consider the algorithm behaviour for the three job execution time estimate models described in Section 5.3.2. The parameters of these models where initialized as follows: E^{ref} refers to execution times depicted in Figure 5.6; *A* is uniformly distributed between $E^{ref}\times 150\%$ and $E^{ref}\times 200\%$; *b* is uniformly distributed between 0 and 1; *F* is uniformly distributed between $E^{ref}\times 5\%$ and $E^{ref}\times 10\%$; and φ is uniformly distributed between $E^{ref}\times 1\%$ and $E^{ref}\times 2\%$. Additionally, the maximum value of E^{est} for the exponentially increasing underestimate model was set equal to amplitude *A*.

To simplify comparison between different models, we assume that input, output, and checkpointing data of all task are equally sized and amount to 1 GB. A checkpointing delay of 2 s is also similar for all tasks.

In all simulation scenarios the system performance is observed during 24 hours of simulated time.

5.6.2 Static versus Dynamic Algorithm

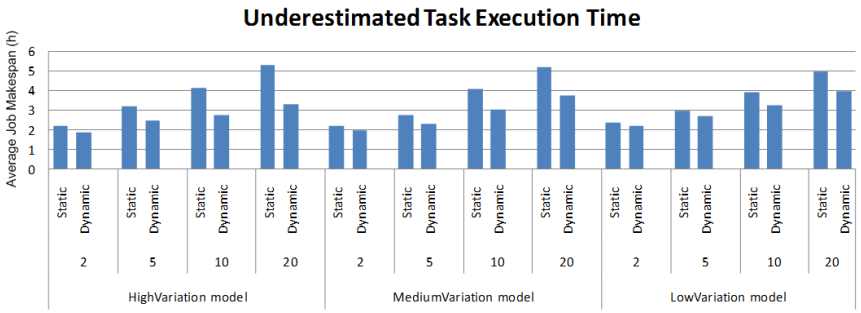
First of all we compare the performance of the originally proposed dynamic algorithm (i.e. disabling the optimizations discussed in Section 5.5) against the performance of a static scheduling approach. The considered static approach works similar to the algorithm discussed in Section 5.4.1, except that it omits the rescheduling phase (Step 2 of the algorithm). It means that the static approach is not able to react on run-time system changes and bases its decisions solely on the information available at the moment a task is scheduled.

This time we assume that an initial rough estimate of the total task execution time is available *a priori*. The initial estimate provided is a random sample of *overestimate*, *underestimate* and *fluctuating* job length variation models, whose parameters were discussed in the previous section. In [34] similar simulations are performed without initial information on task execution time. In the latter case tasks are scheduled randomly, showing, however, analogueous algorithm performance tendencies.

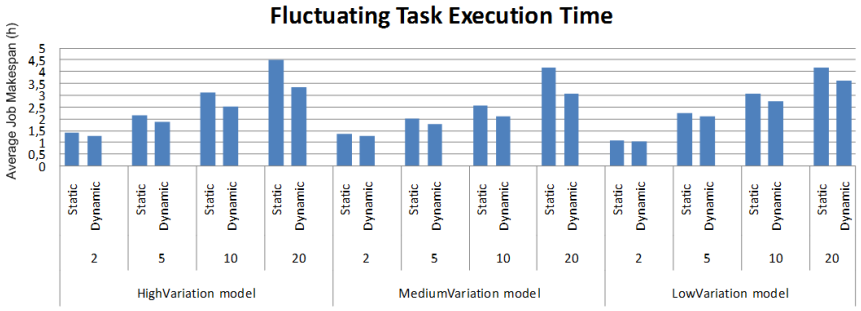
In Figure 5.7 the average makespan of workflows executed by the static and the dynamic algorithms is shown. The results are depicted for jobs with strong, medium and low variations between execution times of the included *parallel* tasks. Also the effect of a varying degree of parallelism within a job is observed, by considering *parallel sets* containing 2, 5, 10 and 20 tasks.

From Figure 5.7 can be concluded that, in general, the dynamic algorithm performs better than its static equivalent. However, the improvement depends on the rescheduling overhead and of course of the execution time predictions. For instance, jobs with *underestimated* execution times, seem to benefit the most from the rescheduling mechanism, which can be explained by the fact that in this case we deal with tasks with relatively long run-times. On the other hand, tasks with *overestimated* execution times have the least profit from rescheduling, since the rescheduling overhead is not compensated by the speed up of the balanced execution of short tasks. In the worst case (low degree of parallelism of *overestimated* tasks), the static approach is even more profitable than the dynamic one. Finally, the computational gain for *fluctuating* tasks is somewhere in between the computational gains of tasks following the *underestimated* and *overestimated* execution time prediction models.

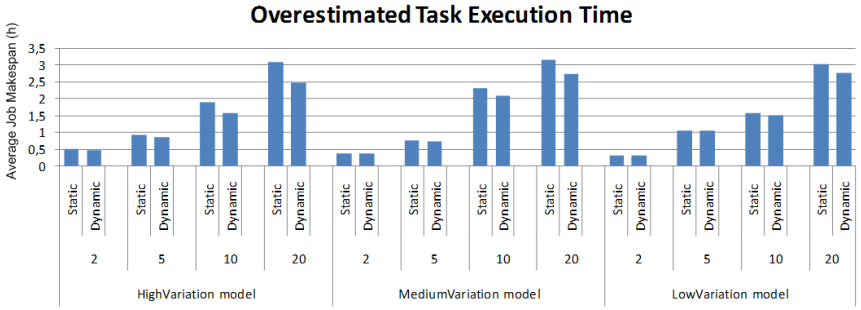
Regarding task length variations and the degree of parallelism within jobs, we can conclude that the more tasks are running within *parallel sets* and the larger is their execution time difference, the more performance gain is achieved with the dynamic algorithm. For example, in the best case of 20 parallel tasks with



(a)



(b)



(c)

Figure 5.7: Average makespan of processed workflows for dynamic and static scheduling approaches for (a) underestimated, (b) fluctuating and (c) overestimated task execution time prediction models. Different average number of tasks within a PS (2, 5, 10, 20) as well as varying load variation models (HighVariation, MediumVariation, LowVariation) are considered.

underestimated strongly varying execution times, the dynamic algorithm reduces job makespan by about 40%, compared to the static algorithm. For the same task execution time prediction model but for low degree of parallelism and low difference in execution times, the utilization of the dynamic algorithm improves the makespan only by 8%.

Table 5.3 summarizes once again the performance improvement realized by the dynamic algorithm, compared to the static approach operating under the same circumstances. We can see clearly that only in the worst case scenario, when relatively short jobs (*i.e.* overestimated prediction), with low degree of parallelism and low variation in task lengths, are scheduled, the static algorithm results in better performance.

5.6.3 Original versus Improved Dynamic Algorithm

In this section we discuss the effect the optimizations presented in Section 5.5 have on the dynamic algorithm performance. We compare *useful workload* executed by different versions of the algorithm. The term *useful workload* refers to the fraction of workload that belongs to tasks successfully terminated within the observed time interval, versus total processed workload. As a use case we choose the *fluctuating* execution time prediction model with tasks with highly varying actual execution times. The model parameters remain unmodified since the previous section.

5.6.3.1 Total Remaining Execution Time Monitoring

Figure 5.8 shows the percentage of *useful workload* processed by the dynamic algorithm, when initialized with different values of RE^{min}. For the grid at hand, the best performance was achieved for RE^{min}=4 min, with up to 15% more *useful workload* processed compared to the original algorithm, with no RE^{min} limit. This gain is mainly the result of the reduced number of checkpoints and migrations performed per task but also of the faster processing of the *parallel* tasks with sufficiently long remaining execution times. The latter have larger choice of the available CRs.

The results shown suggest high sensitivity of the algorithm to relatively small changes in RE^{min}. This can be explained by the choice of the *fluctuating* model where even a small RE^{min} means migration omission at an early stage of task execution. For the *overestimated* and the *underestimated* models larger variance in RE^{min} is necessary to get significant performance difference, since overly small values of RE^{min} have no or little effect due to their rare occurrence. The optimal RE^{min} value should be determined experimentally for each particular grid environment and it depends mainly on application rescheduling overheads (size of inputs/checkpoints, complexity of checkpointing algorithm), execution time prediction evolution, grid load and frequency with which the rescheduling algorithm

	High Variation				Medium Variation				Low Variation			
	2	**5**	**10**	**20**	**2**	**5**	**10**	**20**	**2**	**5**	**10**	**20**
Underestimated Model	+14%	+22%	+34%	+38%	+10%	+15%	+26%	+28%	+8%	+10%	+17%	+20%
Fluctuating Model	+10%	+12%	+20%	+26%	+7%	+10%	+17%	+26%	+5%	+7%	+11%	+13%
Overestimated Model	+4%	+9%	+16%	+20%	+1%	+7%	+10%	+13%	-8%	-1%	+5%	+9%

Table 5.3: Workflow makespan reduction achieved with the dynamic algorithm, compated to the static approach.

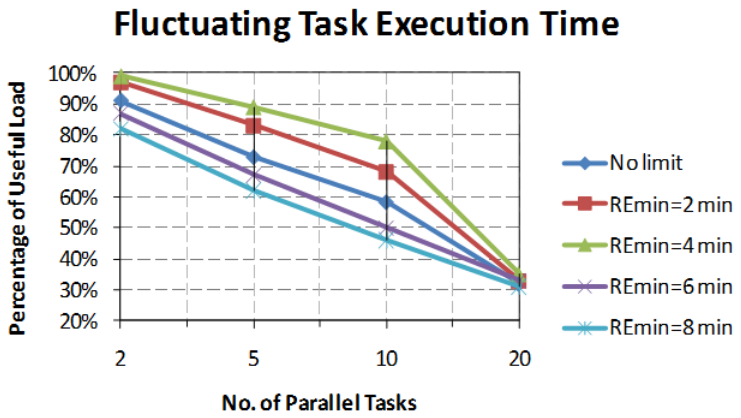


Figure 5.8: Useful load processed for fluctuating execution time estimate model with different values of minimum remaining execution time limit.

is activated.

Important to mention is that despite the fact that Figure 5.8 suggests the largest performance improvement for jobs with limited number of parallel tasks within a PS, also the reduction of the processing time for jobs with large PSs is observed. However, when the degree of parallelism gets high, compared to the number of available resources, the slight grid performance improvement due to checkpoint and rescheduling omission is insufficient to make a difference in the number of processed jobs.

5.6.3.2 Migration Profit Prediction

The γ-based approach is similar to the approach discussed in the previous section and thus it can be used in combination or as an alternative to RE^{min}.

Figure 5.9 shows *useful workload* processed by the dynamic algorithm initialized with different γ-values. We again can observe an improvement of the algorithm performance for small values of γ, due to the rescheduling procedure omissions. However, when γ gets larger and useful migrations get skipped, performance degradation continues until the performance matches the performance of the static approach.

In general, the utilization of γ leads to smaller variations in processed workload compared to the case when RE^{min} is defined. This is the result of the fact that in the first case we only take into the account the performance improvement of tasks within a PS, while in the second case we consider the remaining execution time of PS-tasks as well as tasks chosen for cancellation (see Section 5.4.1.2). Therefore,

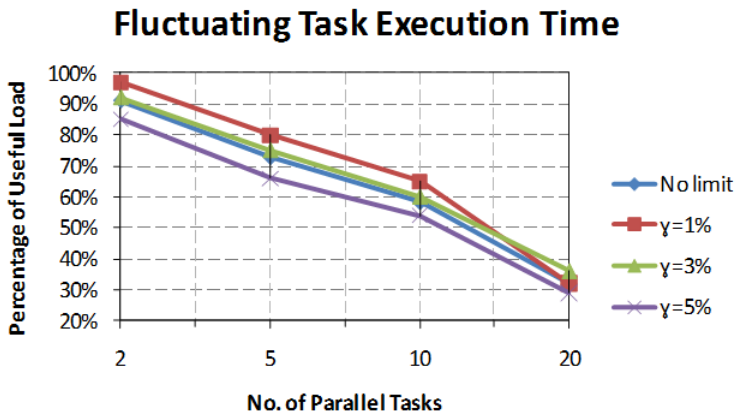


Figure 5.9: Useful load processed for fluctuating execution time estimate model with different values of computational gain limit.

RE^{min} also addresses the overhead related to the restart of canceled tasks.

5.6.3.3 Oscillation Pattern Detection

Taking into account execution time prediction oscillations leads to small additional computational gain (see Figure 5.10). The exact performance improvement depends on oscillation pattern and on the overhead of the rescheduling procedure, however, in general the average execution time prediction based scheduling has shown to be advantageous only when the rescheduling overhead is relatively high.

5.6.3.4 Resource Grouping

Finally, Figure 5.11 shows useful workload processed by the algorithm when resources are grouped using different limits (θ) for the number of CRs within a resource category. The original algorithm performance is indicated by grouping containing a single resource ($\theta = 1$).

The simulation results show that combining up to 4 resources together gives only slight performance degradation, since in our case we manage to include closely collocated CRs with similar execution speeds within a single group. Further reduction of resource categories leads, however, to more significant performance decrease. For instance, in case of $\theta = 16$ the algorithm performs worse than its static equivalent. Also, when PSs are relatively large, the effect of grouping is more distinct, because a single job loss has a larger effect on the overall system performance.

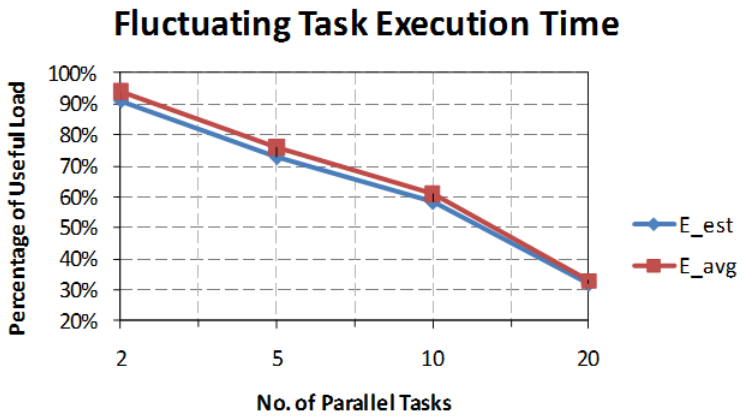


Figure 5.10: Useful load processed for fluctuating execution time estimate model using estimated execution time and average of estimated execution times.

The optimal grouping is largely influenced by the variation in resource capacities and distribution of resources over the network.

5.7 Conclusion

This chapter addresses the issue of scheduling tasks with input interdependencies and unknown execution time within widely distributed grid environments. Tasks are considered for which the execution progress can be monitored at run-time.

We propose a dynamic scheduling algorithm that, based on the periodically collected task progress information and system status updates, constructs total execution time predictions. The latter are used by the algorithm to reschedule already running *dependent* tasks with the objective to reduce the overall application makespan by finding a good balance between rescheduling overhead and efficient utilization of the available resources. The algorithm evaluation within a grid simulation environment shows a performance improvement of up to 35%, compared to the static version of the algorithm.

Despite good performance results, the proposed dynamic algorithm can be subject to further optimizations. The latter mainly address rescheduling overhead reduction and shrinking the search space for the job-resource matchmaking procedure. The following optimizations are considered: migration omission for tasks with short remaining execution time prediction and for cases when the rescheduling profit is expected to be low; utilization of average execution time prediction values when the prediction history suggests an ossilating behaviour pattern; group-

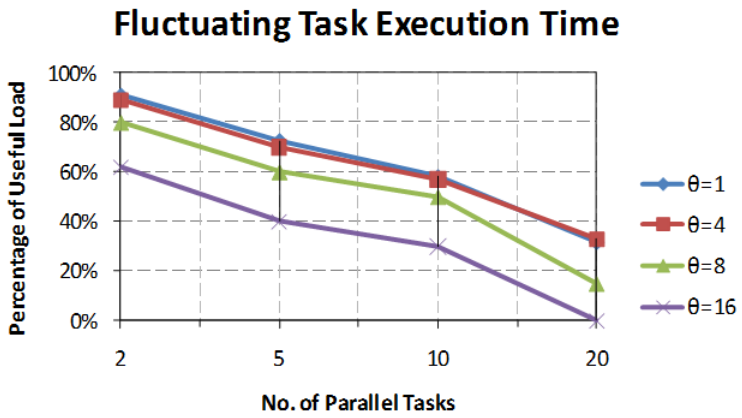


Figure 5.11: Useful load processed for fluctuating execution time estimate model with different granularity of resource grouping.

ing of similar CRs into categories and performing task assignment per categories. Our simulation results show that the suggested approaches lead to performance improvement, varying from 0.5% to 15%.

References

[1] F.D. Sacerdoti, M.J. Katz, M.L. Massie, and D.E. Culler. *Wide Area Cluster Monitoring with Ganglia*. In Proceedings of the IEEE International Conference on Cluster Computing, pages 289 – 298, Kowloon, Hong Kong, China, December 1 – 4 2003.

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. *High-performance, Portable Implementation of the MPI Message Passing Interface Standard*. Parallel Computing, 22(6):789 – 828, 1996.

[3] F. Claeys, D.J.W. De Pauw, L. Benedetti, I. Nopens, and P.A. Vanrolleghem. *Tornado: a Versatile and Efficient Modelling & Virtual Experimentation Kernel for Water Quality Systems Applications*. In Proceedings of the International Environmental Modelling and Software Conference (iEMSs), Burlington, Vermont, USA, July 9–13 2006.

[4] U. Lublin and D.G Feitelson. *The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs*. Parallel & Distributed Computing, 63(11):1105 – 1122, November 1992. 2003.

[5] A. Mandal, K. Kennedy, C. Koelbel, Marin G., J. Mellor-Crummey, B. Liu, and L. Johnsson. *Scheduling Strategies for Mapping Application Workflows onto the Grid*. In Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing, pages 125 – 134, Research Triangle Park, NC, USA, July 24 – 27 2005.

[6] M. Wieczorek, R. Prodan, and T. Fahringer. *Scheduling of Scientific Workflows in the ASKALON Grid Environment*. ACM SIGMOD Record, 34(3):56 – 62, September 2005.

[7] F. Dong and G.A. Selim. *PFAS: A Resource-Performance-Fluctuation-Aware Workflow Scheduling Algorithm for Grid Computing*. In Proceedings of the International Parallel and Distributed Processing Symposium, pages 1 – 9, Long Beach, CA, USA, March 26 – 30 2007.

[8] M. Rahman, S. Venugopal, and R. Buyya. *A Dynamic Critical Path Algorithm for Scheduling Scientific Workflow Applications on Global Grids*. In Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing, pages 35 – 42, Bangalore, India, December 10 – 13 2007.

[9] Y. Yan and B. Chapman. *Scientific workflow scheduling in computational grids - Planning, reservation, and data/network-awareness*. In Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, pages 18 – 25, Austin, TX, USA, September 19 – 21 2007.

[10] Z. Yu and W. Shi. *An Adaptive Rescheduling Strategy for Grid Workflow Applications*. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium, pages 1 – 8, Long Beach, CA, USA, March 26 – 30 2007.

[11] Z. Ding, X. Wei, Y. Zhu, Y. Yuan, W.W. Li, and O. Tatebe. *Implementation of the Grid Workflow Scheduling for Data Intensive Applications as Scheduling Plugins*. In Proceedings of the 2nd International Conference on Future Generation Communication and Networking, pages 14 – 20, Sanya, Hainan Island, China, December 13 – 15 2008.

[12] Y. Zhang, C. Koelbel, and K. Cooper. *Cluster-Based Hybrid Scheduling Mechanisms for Workflow Applications on the Grid*. In Proceedings of the 4th IEEE International Conference on eScience, pages 390 – 391, Indianapolis, Indiana, USA, December 7 – 12 2008.

[13] T. Dornemann, E. Juhnke, and B. Freisleben. *On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud*. In Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pages 140 – 147, Shanghai, China, May 18 – 21 2009.

[14] Y.C. Lee, R. Subrata, and A.Y. Zomaya. *On the Performance of a Dual-Objective Optimization Model for Workflow Applications on Grid Platforms*. IEEE Transactions on Parallel and Distributed Systems, 20(9):1273 – 1284, September 2009.

[15] J. Yu, R. Buyya, and C.K. Tham. *Cost-based Scheduling of Scientific Workflow Applications on Utility Grids*. In Proceedings of the 1st International Conference on e-Science and Grid Computing, pages 139 – 147, Melbourne, Australia, December 5 – 8 2005.

[16] Y. Yuan, T. Yu, F. Xiong, and M. Fang. *QoS-based dynamic scheduling for manufacturing grid workflow*. In Proceedings of the 9th International Conference on Computer Supported Cooperative Work and Design, pages 1123 – 1128, Coventry University, UK, May 24 – 26 2005.

[17] Y. Yuan, X. Li, and Q. Wang. *An iterative heuristic for scheduling grid workflows with budget constraints*. In Proceedings of the 10th International Conference on Computer Supported Cooperative Work in Design, pages 1 – 6, Southeast University, Nanjing, China, May 3 – 5 2006.

[18] A. Afzal, J. Darlington, and A.S. McGough. *Stochastic Workflow Scheduling with QoS Guarantees in Grid Computing Environments*. In Proceedings of the 5th International Conference on Grid and Cooperative Computing, pages 185 – 194, Changsha, Hunan, China, October 21 – 23 2006.

[19] N. Ranaldo and E. Zimeo. *Time and Cost-Driven Scheduling of Data Parallel Tasks in Grid Workflows*. IEEE System Journal, 3(1):104 – 120, 2009.

[20] H.M. Fard and H. Deldari. *An Economic Approach for Scheduling Dependent Tasks in Grid Computing*. In Proceedings of the 11th IEEE International Conference on Computational Science and Engineering Workshops, pages 71–76, São Paulo, Brazil, July 16 – 18 2008.

[21] A. Simion, D. Sbirlea, F. Pop, and V. Cristea. *Dynamic Scheduling Algorithms for Workflow Applications in Grid Environment*. In Proceedings of 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Timisoara, Romania, September 26 – 29 2009.

[22] D. Changsong, H. Zhoujun, H. Zhigang, and L. Xi. *A Distributed Workflow Management System Model and its Scheduling Algorithm*. In Proceedings of Japan-China Joint Workshop on Frontier of Computer Science and Technology, Nagasaki, Japan, December 27 – 28 2008.

[23] B.J. Lafreniere and A.C. Sodan. *ScoPred – Scalable User-Directed Performance Prediction Using Complexity Modeling and Historical Data*. In Proceedings of the 11th Workshop on Job Scheduling Strategies for Parallel Processing, pages 62 – 90, Cambridge, MA, June 19 2005.

[24] F. Nadeem and T. Fahringer. *Using Templates to Predict Execution Time of Scientific Workflow Applications in the Grid*. In Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pages 316 – 323, Shanghai, China, May 18 – 21 2009.

[25] R. Prodan and T. Fahringer. *Overhead Analysis of Scientific Workflows in Grid Environments*. IEEE Transactions on Parallel and Distributed Systems, 19(3):378 – 393, March 2008.

[26] H. Casanova and L. Marchal. *A Network Model for Simulation of Grid Application*. Technical report, École Normale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme, 2002.

[27] F.H.A. Claeys. *A Generic Software Framework for Modelling and Virtual Experimentation with Complex Environmental Systems*. Phd thesis, Ghent University, Department of Applied Mathematics, Biometrics and Process Control, Coupure Links 653, B-9000 Gent, Belgium, January 2008.

[28] K. Schwarz, P. Blaha, and G.K.H. Madsen. *Electronic Structure Calculations of Solids Using the Wien2k Package Material Sciences*. Computer Physics Communications, 147(71), 2002.

[29] M. Chtepen, F.H.A. Claeys, B. Dhoedt, F. De Turck, P.A. Vanrolleghem, and P. Demeester. *Dynamic Scheduling of Computationally Intensive Applications on Unreliable Infrastructures*. In Proceedings of the 2nd European Modeling and Simulation Symposium (EMSS '06), Barcelona, Spain, October 4 – 6 2006.

[30] R. Buyya and M. Murshed. *GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing*. The Journal of Concurrency and Computation: Practice and Experience (CCPE), 14(13–15), November – December 2002. Wiley Press.

[31] A. Legrand, L. Marchal, and H. Casanova. *Scheduling Distributed Applications: the SimGrid Simulation Framework*. In Proceedings of the 3st International Symposium on Cluster Computing and the Grid (CCGrid), Tokyo, Japan, May 12–15 2003.

[32] P. Thysebaert, B. Volckaert, F. De Turck, B. Dhoedt, and P. Demeester. *Evaluation of Grid Scheduling Strategies through NSGrid: a Network-Aware Grid Simulator*. Special issue on grid computing of the Journal of Neural, Parallel and Scientific Computations, 12(3):353–378, 2004.

[33] UGent. *BeGrid UGent*. http://begrid.atlantis.ugent.be/.

[34] M. Chtepen, F.H.A. Claeys, B. Dhoedt, F. De Turck, P.A. Vanrolleghem, and P. Demeester. *Scheduling of Dependent Grid Jobs in Absence of Exact Job Length Information*. In Proceedings of the 4th IEEE/IFIP International Workshop on End-to-end Virtualization and Grid Management, Samos Island, Greece, September 22 – 26 2008.

6

On-line Execution Time Prediction for Computationally Intensive Applications with Periodic Progress Updates

M. Chtepen, F.H.A. Claeys, B. Dhoedt, F. De Turck, J. Fostier, P. Demeester and P.A. Vanrolleghem

Submitted to *The Journal of Supercomputing*.

⋆ ⋆ ⋆

The effectiveness of distributed execution of computationally intensive applications (jobs) largely depends on the quality of the applied scheduling approach. However, most of the existing non-trivial scheduling algorithms rely on prior knowledge or on prediction of application parameters, such as execution time, size of input and output, dependencies, etc., to assign applications to the available computational resources. A major issue is that these parameters are hard to determine in advance, especially if the end user does not possess an extensive history of previous application runs.

We address the issue of job execution time prediction, mentioned in Chapter 5, by proposing an on-line prediction method for applications for which execution progress can be collected at run-time. In the previous chapter, the total job execution time is predicted using extrapolation of the last progress update. However, the

predictions achieved by extrapolation are far from precise and often vary over time as a result of changing application dynamics and varying resource load. Therefore, to compute the actual job execution time we propose to match a number of predefined prediction evolution models against the consecutive extrapolations, by adopting nonlinear curve-fitting. The "best-fit" coefficients allow for more accurate execution time prediction.

The predictions made are used to enhance a dynamic scheduling algorithm for workflows introduced in the previous chapter. The scheduling algorithm is run with and without curve-fitting, showing a performance improvement of up to 15% in the former case.

6.1 Introduction

When dealing with application complexity and long execution times, one often thinks of distributed solutions such as clusters and grids. However, the benefit of distributed approaches largely depends on the scheduling strategy applied. Most of the currently existing schedulers for distributed systems require sufficiently accurate information to be provided on the applications scheduled and on the available resources, to perform effective job-resource matchmaking. Unfortunately, this information is hard to obtain in advance, due to high job diversity, large variation in input parameters and the dynamic nature of distributed resources.

One of the major issues in the domain of distributed computing is considered to be prediction of application run-times. Taking into account the diversity of the existing applications, it seems extremely difficult to define a general solution to the problem. Therefore, in this chapter we concentrate on the category of applications for which the execution progress can be monitored at run-time. We propose a dynamic prediction mechanism that iteratively refines estimates of job execution time based on periodic run-time updates on the job progress.

In concreto, there exists a broad group of applications for which execution progress can either be collected periodically or at particular timepoints during the application execution. Examples of such applications are simulations with a total simulated time T^{total} that is known *a priori*. If, for instance, the current simulated time (t) can be collected at run time, we can predict the total execution time of our job J (E_J^{est}) using extrapolation: $E_J^{est} = T_J * (T^{total}/t)$, where T_J is the processing time of J thus far. Other examples are applications consisting of a number of consecutive runs. When the total and the current number of runs are known, we can extrapolate as above, to predict E_J^{est}. The problem of this simple approach is that the predicted execution time E_J^{est} can strongly vary over time, depending on system dynamics, complexity of individual job runs and changes in resource load. To address this dynamic behaviour of applications and distributed resources, we modify E_J^{est} at run-time, taking into account the previous job execution time

predictions.

We make the realistic assumption that E_J^{est} converges over time to a certain end point, which means that the longer a job is running, the closer E_J^{est} converges to the actual job execution time (E_J^{act}). Therefore, we can match the course of E_J^{est} against a number of *prediction evolution* models, determined using historical information on previous application runs. The matchmaking is realized using a curve-fitting optimization procedure. Parameters determined by the optimization provide an accurate estimate of the convergence point and thus of the total execution time.

Obviously, only a dynamic scheduling approach can benefit from the proposed prediction mechanism. A scheduler should be able to assign arriving jobs randomly to the available resources and to reschedule them at run time as more information on individual job progress becomes available. Therefore, to evaluate the performance of our prediction method, the latter is incorporated into a dynamic scheduling algorithm for workflow applications that is introduced in our previous publication [1]. The algorithm was simulated in a grid simulation environment (DSiDE [2]), using realistic workload derived from a modeling and simulation tool for environmental systems (Tornado [3]).

The remainder of this chapter is structured as follows: Section 6.2 introduces related work; Section 6.3 describes the job/task execution time prediction method proposed; the use case scenario utilized for evaluation of the prediction method is discussed in Section 6.4; simulation results can be found in Section 6.5; and, finally, Section 6.6 concludes the chapter.

6.2 Related Work

Currently existing approaches for estimating execution times of jobs running in distributed environments can be subdivided into two main categories: application component performance modeling and historical prediction.

Application component performance modeling considers the number and the complexity of instructions executed for particular input parameters. A concrete example of this approach can be found in [4], where a job is run initially with several small-size input problems. The computational complexity for each run is determined as a function of the number of floating point operations performed and the memory access pattern. After the data collection phase, least square curve-fitting is applied on the collected data to make prediction for a specific input data set. The main disadvantage of this approach is that it operates at a fine-grained instruction level and is, therefore, only applicable to small, deterministic applications with a limited number of input parameter combinations. A slightly different mechanism is proposed in [5]. Here the execution time prediction is taken to a next level of abstraction, by identifying a number of primitive routines performed

by a job. The total execution time prediction is derived from the job performance within the routines. Other application model-based prediction solutions are discussed in [6] and [7]. In [6] dynamic models of workload evolutions are designed to predict the execution time of non-deterministic bulk synchronous computations on multiprocessors. In [7] a modeling approach to estimating the execution time of long-running scientific applications is presented. The approach is based on the observation of resource usage behaviour of a job and job profiling.

In general, it can be concluded that while exhaustive profiling within application performance modeling provides for very accurate estimates, correct application models are hard or sometimes even impossible to obtain. Furthermore, the approach yields insufficient insights on the impact of input data changes on application execution.

On the other hand, the historical prediction method that utilizes sets of past observations to predict execution times, seems to be more performant and more generally applicable. Therefore, it is frequently applied within recent research projects. For instance, in [8] the ScoPred performance predictor is discussed, which applies multiple linear regression using rough estimates of application execution time provided by the end user and historical application execution data to predict the execution times. Other regression-based methods are described in [9] and [10]. Here regression models and filtering techniques are applied on a subset of previous application runs in order to discover the relationships between variables that affect the run times of applications (e.g., application input, resource capacities). In [11] and [12] *similarity template*-based approaches are proposed. A *similarity template* refers to a set of selected attributes that allow to determine similarities between application runs. In [11] matchmaking of templates is supervised by an expert user, who is supposed to indicate the relevance of each attribute for a particular application. In [12], on the other hand, similarity distance calculations are performed on attributes in a predefined order. Hereby, the similarity calculation of the second most relevant attribute will occur only for those cases that get high similarity for the most relevant attribute.

The main disadvantage of the historical approach is that a large number of historical records must be stored before matchmaking of a job against the available records can provide a sufficiently accurate estimate. However, the more records are stored, the longer the matchmaking procedure takes.

The approach proposed in this chapter can be classified as high level application performance modeling, where the possible models of job execution progress evolution are provided by end users in advance. The advantage of our approach is that it requires a relatively limited number of *prediction evolution* models to deliver acceptable prediction accuracy.

6.3 Prediction Algorithm Description

The algorithm proposed is primarily designed for dynamic schedulers assigning jobs with *a priori* unknown execution times within dynamic distributed environments. The mechanism is implemented as an independent module that can be plugged into a scheduler. The idea is that the scheduler periodically consults the prediction algorithm to determine a new job execution time estimate, based on the earlier collected job progress history. The remainder of this chapter presents the algorithm pseudo code and gives explanation on consecutive steps.

The functionality of the proposed algorithm can be formalized as follows:

Input: *Job name*: J,
Set of (estimate, progress collection time)-pairs: $Hist = \{(E_1, T_1), ..., (E_n, T_n)\}$,
Set of initial parameter values: $Init = \{i_1, ..., i_m\}$,
Set of prediction evolution functions: $Func = \{(\psi_1, ..., \psi_k\}$
Output: *Job execution time estimate*: E_J^{est}

1: **if** $Size(Hist) \equiv GetPreviousEstimatesNo(J)$ **then**
2: $E_J^{est} \Leftarrow E_n$
3: **else**
4: $SetPreviousEstimatesNo(J, Size(Hist))$
5: **if** $Size(Hist) \equiv 1$ **then**
6: $E_J^{est} \Leftarrow E_1$
7: **else**
8: **for all** $\psi_j \in Func$ **do**
9: $[X_{\psi_j}, ResNorm_{\psi_j}] \Leftarrow MatchCurve(\psi_j, Init, Hist)$
10: **end for**
11: $\psi_j \Leftarrow MinResNormGet(ResNorm)$
12: $E_J^{est} \Leftarrow CalculateLimit(\psi_j, X_{\psi_j})$
13: **end if**
14: **end if**

The input to the algorithm consists of the following parameters: the name of a job (J), for which a new estimate of the execution time is required; the prediction history $Hist$, containing pairs of execution time predictions E, computed by extrapolation of consecutive progress measurements, and progress collection timestamps T; and a set $Init$ of initial parameter values (i). The initial parameters serve to initialize the optimization performed in the scope of curve-fitting. The parameters within $Init$ are provided by end users, which are presumed to possess sufficient application knowledge to provide for the appropriate initial values. A good choice of the latter is highly important for the accuracy of the prediction mechanism, since it avoids the optimization ending in a local optimum. Also the set $Func$ of functions ψ is provided to the algorithm. Each function ψ describes

a possible scenario for an execution time estimates evolution over time. The historical input-data (E) is matched against the available functions to provide a new estimate E_J^{est}, which is the output of the algorithm.

Before proceeding with calculating E_J^{est}, the algorithm first checks whether new information on job progress has become available since the last algorithm run. If the latter is not the case, the previous value of E_J^{est} still applies (see Lines 1 – 2). On the other hand, when the number of extrapolated estimates increases ($Size(Hist)$), this number is saved for the next run and the algorithm proceeds with computing the new E_J^{est} (see Lines 3 – 14).

Obviously, it is assumed that the prediction algorithm is called only after at least one progress indication is collected. However, since no curve fitting can be performed for a single point, the algorithm simply returns the initial estimate value E_1 (see Lines 5 – 6). When the set $Hist$ contains multiple estimates, for each predefined function ψ the curve-fitting optimization procedure *MatchCurve* is called (see Lines 7 – 9). The *MatchCurve* method takes as arguments a function ψ, the initial parameter values and the execution time estimate data points, together with the estimate timestamps. The outputs of the method are two vectors: X contains the parameter values that best fit function $\psi_j(X, T)$ to the data $Hist$; and *ResNorm*, which represents the residual norm, used by the prediction algorithm to determine the function ψ_j that best fits the provided input data. *ResNorm* is calculated as the squared 2-norm of the residuals (see Formula 6.1), which means the parameter depicts the squared difference between the optimized function and the input data. Clearly, the smaller the difference, the better the curves fit.

$$ResNorm_{\psi_j} = \sum_k (\psi_j(Init, T_k) - E_k)^2 \tag{6.1}$$

Finally, the limit of the function with the minimum *ResNorm* is calculated, which provides us with the a new execution time prediction (see Lines 11 – 12). As was mentioned previously, we assume that the longer a job is running, the closer its execution time prediction gets to the real execution time value. This means that the function representing the execution time evolution converges over time to a certain limit-value. Since there are always only a limited number of functions provided, the limit expression can easily be determined analytically by an end user and provided to the algorithm together with the *prediction evolution* functions ($Func$). Afterwards, the predictions can be calculated by substituting the X-parameters into the limit expression of the ψ_j-function.

6.4 Use Case Description

To evaluate the performance of the proposed prediction algorithm we integrate the latter into the dynamic workflow scheduler introduced in the previous chapter [1].

The scheduler is implemented in an existing grid simulation environment, called DSiDE [2], which allows for easy modeling and monitoring of dynamic resource and application behaviour. To get an accurate indication on the overhead periodic prediction and rescheduling introduce in distributed environments, a realistic medium-sized grid model and a workload model deduced from a real-world application are considered.

In the remainder of this section, the workload and the grid models, together with the utilized dynamic scheduling approach are discussed in more details.

6.4.1 Workload Model

The performance of the proposed algorithm is simulated using a workload model derived from Tornado [3], an existing application for modeling and virtual experimentation with complex environmental systems. Tornado is particularly interesting as a use case since it generates jobs with strongly varying properties in terms of job execution times, mutual dependencies, size of input/output data, *etc*.

In this chapter we consider Tornado jobs composed of tasks with input dependencies. It means that some tasks require inputs generated by other tasks, before they can proceed with their execution. This type of dependency can significantly benefit from distributed execution, compared, for instance, to MPI-based [13] dependencies, since it does not require extensive communication between tasks at run-time.

In general, a Tornado job with input dependent tasks can be represented as a DAG of the form shown in Figure 6.1. As depicted on the figure, each job contains a single *initial task*, which generates inputs for one or several *dependent tasks*, which in turn generate inputs for their *dependent tasks*, *etc*. The procedure continues until the final level of the dependency hierarchy is reached, where a single *final task* produces job results. An example of such a dependency structure within Tornado is the Scenario Analysis experiment: the *initial task* determines different parameter values, input variable values and/or initial conditions; afterwards, individual simulation experiments (*dependent tasks*) are run with each combination of inputs and during each run the simulated trajectories of a number of selected quantities are saved; the *final task* is executed to compute a variety of objective values.

Due to a large diversity of possible inputs for Tornado experiments, it is hard to predict the execution time of an experiment in advance. However, for a large group of Tornado jobs the execution progress can be monitored at run time and their total execution time can be predicted using extrapolation:

$$E_J^{est} = \frac{100\% \times T_J \times MIPS_{CR}}{P_J \times n_{CR}} \tag{6.2}$$

where J is a Tornado task running on a distributed computational resource CR;

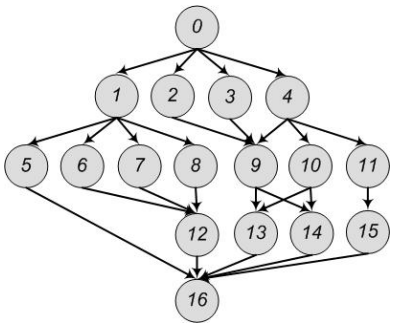


Figure 6.1: Example of a workflow consisting of tasks with input-dependencies, organized into a DAG structure

T_J is the wall clock execution time of J thus far; $MIPS_{CR}$ is the speed of the resource CR; P_J is the percentage of the task J completed within the time period T_J; and n_{CR} is the total number of tasks running on CR. In fact, we compute the execution time prediction on a theoretical reference resource CR^{ref}, having $MIPS_{CR^{ref}} = 1$ and $n_{CR^{ref}} = 1$.

To provide for a realistic model for evolution of execution time estimates, a number of Tornado experiments were observed. From these observations can be concluded that the estimate curves show strongly alternating evolution (see Figure 6.2). However a common tendency can be distilled by defining the following three approximation models:

- The **overestimate model** represents estimates that are gradually decreasing until the *stable state* is reached. In the *stable state* the exact execution time is known and the predictions no longer change. An example of this model is the "Galindo_CL" simulation experiment.
- The **underestimate model** is the opposite of the *overestimate model*. Here the execution time prediction increases until the *stable state*. "BSM1_CL" is an example of this model.
- The **fluctuating model** represents an erratic pattern, whereby predictions oscillate over time. In the case of "Galindo_OL" and "Bamberg" we can talk about the *fluctuating model*.

The above mentioned models can be approximated mathematically by the following exponential functions: $E_J^{est} = E_J^{ref} + r_1 A e^{-bt} \pm r_3, r_1 > 0$ describes the exponentially decreasing *overestimate model*; $E_J^{est} = E_J^{ref} - r_1 A e^{-bt} \pm r_3, r_1 > 0$ represents the exponentially increasing *underestimate model*; and, finally, $E_J^{est} = E_J^{ref} \pm r_1 A e^{-bt} sin(2\pi F t + r_2 \varphi) \pm r_3, r_1 > 0$ represents the *fluctuating model*. In these equations E_J^{ref} is a reference execution time value for the job J that prevents the models from having too short initial execution time estimates; r_1 and

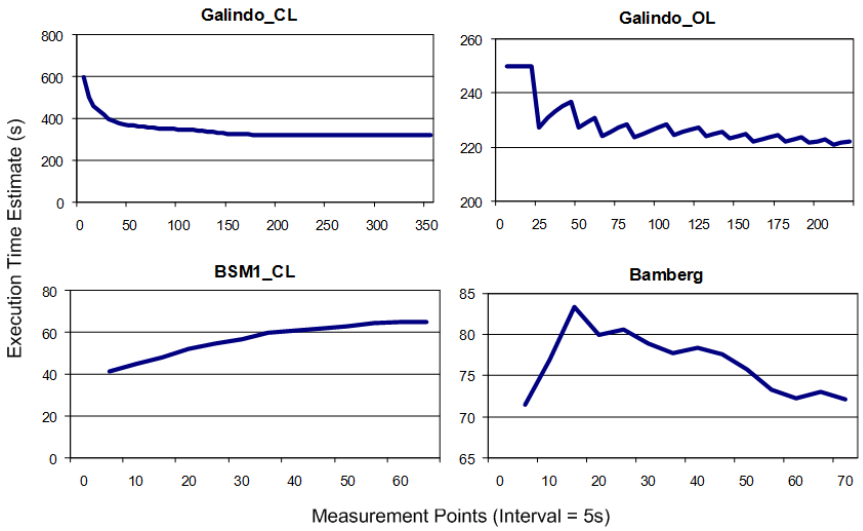


Figure 6.2: Examples of evolution of execution time estimates for the "Galindo_CL", "Galindo_OL", "BSM1_CL" and "Bamberg" simulation experiments.

r_2 are random weight factors that can take values between 0 and 1; A is the amplitude of estimate oscillations; b is a weight factor that determines the speed in the increase/decrease of A over time; F stands for the oscillation curve period; φ represents the oscillation curve phase; and, finally, $r_3 = pA$ is white noise that is defined as a small percentage p of the amplitude. Obviously, the larger p, the more noise is imposed on the model and the more difficult it is to classify the curve. To eliminate/reduce noise, *filtering* or *smoothing* can be applied on the input data before the optimization step.

Another issue is that it is often difficult to distinguish noise from the *oscillating model* pattern. However, we are not really interested in oscillations but rather in the end values, which remain after the oscillations have decayed. We partially address both issues by providing to the prediction algorithm the following two *prediction evolution* functions: $\psi_1 = x_1 + x_2 e^{(-x_3 t)}$ and $\psi_2 = y_1 - y_2 e^{(-y_3 t)}$, where x_j and y_j stand for the parameter values to be determined. To a certain extent, the functions have the effect of a *smoothing* technique, since after sufficiently long task run-times they capture the increasing/decreasing data pattern and manage to eliminate noise and oscillations. The limits of the functions can be analytically determined as x_1 and y_1.

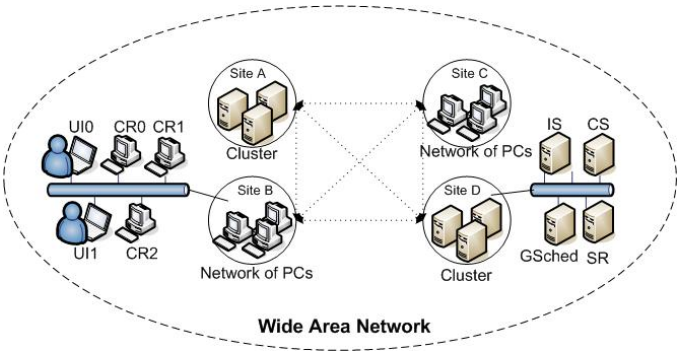


Figure 6.3: Considered grid model: Computational Resource (CR), Grid Scheduler (GS), User Interface (UI), Information Service (IS), Checkpoint Server (CS).

6.4.2 Grid Model

We model a grid environment (see Figure 6.3) consisting of a number of distributed Sites (S), aggregating heterogeneous dedicated Computational Resources (CR).

In practice, resource capacity is a complex quantity, which is influenced by different hardware components. In this work, we use a simplified metric, called MIPS, which is often applied theoretically to compare resource capacities.

Another assumption relates the sharing of CR capacity among simultaneously active tasks. Normally, each application requires different and alternating amounts of hardware resources (CPU, IO bus, *etc.*), but we assume that each task J is allocated an equal share of a resource capacity ($MIPS_J^{CR}$), which is determined using the following equation:

$$MIPS_J^{CR} = \frac{MIPS_{CR}}{n_{CR}}. \tag{6.3}$$

To avoid overzealous partitioning of $MIPS_{CR}$, the number of tasks allowed to run on a CR simultaneously is limited by the boundary n_{CR}^{max}: $n_{CR} \leq n_{CR}^{max}$.

Next to CRs, the considered grid infrastructure includes a number of general services, such as a Grid Scheduler (GS), which maintains a job queue and is responsible for task-resource matchmaking; several distributed User Interfaces (UIs), utilized by end users to submit their applications to the grid; an Information Service (IS) required to collect information on the grid status; a Checkpoint Server (CS) where checkpoints are saved; and a Storage Resource (SR) where output data is transferred after a job execution.

Often, a real-world grid infrastructure contains several SRs. The allocation of input/output data to an appropriate SR is then performed using a certain data scheduling policy. Since data scheduling is out of the scope of this work, we limit the considered grid infrastructure to a single SR.

The performance evaluation of the proposed prediction algorithm would not be accurate if we would not consider different types of overhead, related to the dynamic information collection and the rescheduling process.

The first important source of overhead relates to the grid middleware services: querying IS for dynamic resource/task status, periodic task execution time predictions, (re)schedule computations by GS, and, finally, checkpointing and migration slow down task execution. These types of overhead are taken into account by adding constant delays to our model.

The second significant cause of task slow down originates from network transfers, such as input/output file staging, workflow rescheduling and rollback. In our model all grid sites are interconnected by a Wide Area Network (WAN), while the communication within the sites go by different Local Area Networks (LANs). It is assumed that network transfers within LANs originate exclusively from grid jobs, while WANs are open to external network traffic. Therefore, two different models, described in [14], are used for bandwidth (B) sharing among simultaneous transfers: every data transfer route going through a WAN link gets a small equal share of the total link capacity; while capacities of LAN links are proportionally shared among simultaneous active grid transfers. For simplicity we assume that total link capacities do not change over time and that links are not subject to failure.

6.4.3 Dynamic Scheduling Algorithm

In this work we integrate our execution time prediction module into a dynamic scheduling algorithm introduced in the previous chapter, to observe to what extent the predictions made improve the algorithm performance. The algorithm operates in dynamic grid environments where tasks with input dependencies and unknown execution times (for which, however, periodic progress information can be collected) are run. The algorithm makes use of information services to collect dynamic system updates and applies these updates to (re)schedule dependent tasks. Figure 6.4 gives a brief overview of the different algorithm steps, which are described in more details in the remainder of this section.

The objective of the algorithm is to reduce the execution time of a job (see Figure 6.1) running on a set of distributed heterogeneous resources, by taking into account task interdependencies. Before we proceed, we define the notion of a *parent set* (PS), which is a set of tasks generating input for the same group of *dependent tasks*. For example, in Figure 6.1 tasks $\{0\}$, $\{1\}$, $\{2,3,4\}$, $\{4\}$, $\{6,7,8\}$, $\{9,10\}$, $\{11\}$, $\{5,12,13,14,15\}$ form *parent sets* for respectively tasks 1 – 4, 5 – 8, 9, 10 – 11, 12, 13 – 14, 15 and 16. Clearly, each task in the considered workflow, except for the *initial task*, has a *parent set*. *Parent sets* of different tasks are not necessarily unique and they can overlap, which is the case for the sets $\{2,3,4\}$ and $\{4\}$ in the example above.

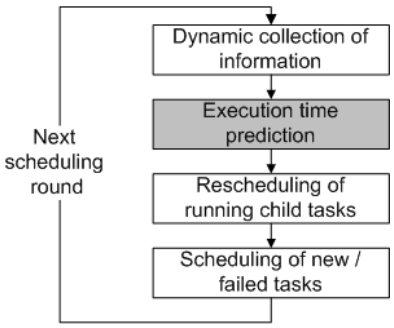


Figure 6.4: Flow of the operation phases of the dynamic scheduling algorithm.

Several issues arise when we are dealing with input-dependency constraints. First of all, if jobs arrive with high frequency into a grid, *initial tasks*, which do not have input dependencies and can be started immediately, occupy the available resources to a large extent. This leads to a delay in the execution of *dependent tasks* and thus prolongs the job execution as a whole. Secondly, not all tasks within a *parent set* are computationally equally intensive, which means that some tasks may finish much faster than others, when executed on resources with similar capacity. The imbalanced task execution, however, is not advantageous because the output of a whole *parent set* is required to proceed with the execution of its *dependent tasks*.

Since we are usually not interested in partial results but only in the results produced by *final tasks*, the algorithm tries to reduce the execution time of a job at the cost of possibly slower execution of individual tasks. The idea behind the algorithm is to give the processing of *dependent tasks* a higher priority than the execution of *initial tasks*. In fact, the latter are scheduled only when no *dependent tasks* are waiting for the execution. Furthermore, the execution of *parent sets* is balanced by scheduling the computationally most intensive tasks within a set to the fastest available resources, leaving slow resources to short tasks. Our optimization heuristics can formally be described by the following equations:

$$\min_{\forall PS \in \mathcal{W}} (\max_{\forall J_i \in PS} \{E^{est}_{J_i}\}) \tag{6.4}$$

$$\forall PS \in \mathcal{W} : \min_{\forall J_i, J_k \in PS} \mid E^{est}_{J_i} - E^{est}_{J_k} \mid \tag{6.5}$$

These equations mean that the maximum execution time prediction ($\max_{\forall J_i \in PS}$ $\{E^{est}_{J_i}\}$) as well as the difference of task execution time predictions ($\mid E^{est}_{J_i} - E^{est}_{J_k} \mid, \forall J_i, J_k \in PS$) within each *parent set* should be minimized consecutively. Clearly, to satisfy these criteria and to provide an appropriate schedule, the algorithm requires a good task execution time estimate mechanism. The better the

provided estimate, the less rescheduling needs to be performed on each system / task dynamic information update.

In concreto, the algorithm of Figure 6.4 proceeds as follows:

- **Collection of dynamic information**. The information on resource load and availability, as well as the information on job status and progress of running jobs is collected. Important to mention is that the data collected can be outdated, depending on the length of the interval used by the IS to query the grid.
- **Execution time prediction**. In this step, the execution time predictions of tasks within *running PSs* (RPSs) are (re)computed, based on updates in task progress and resource status. By the term *running PS* we understand a PS that exclusively contains finished tasks and tasks actually running on active grid resources, but no waiting tasks.
- **Rescheduling of Running PSs**. Tasks within RPSs are reassigned to balance their predicted execution times: the longest task is assigned to the fastest available resource (minimizing maximum execution time), while other tasks are assigned such that their execution times are as close as possible to the execution time of the longest task within the RPS, without actually exceeding it (minimizing processing time difference). This means that the shortest tasks are migrated to slowest resources, leaving the fastest resources to the tasks requiring fast execution.
- **Scheduling of idle tasks**. The *parent sets* containing idle tasks are assigned to the resources remaining after the rescheduling step. The PSs are processed in the order of their arrival into the GS-queue. For some applications we may possess an initial estimate of the total tasks execution time. In this case the scheduler proceeds as described in the previous step. Otherwise, tasks are assigned randomly.

  More detailed information on the different steps of the algorithm can be found in [1].

6.5 Algorithm Performance Evaluation

To measure the benefit of the proposed prediction algorithm for complex jobs with unknown execution times, a number of simulation experiments were performed in the DSiDE simulation environment. In this section we describe the simulation scenario parameters (see Table 6.1), together with the performance results.

6.5.1 Simulation Experiment Description

A grid consisting of 4 computational sites, with 32 CRs each, is observed during 24 hours of simulated time. Each CR within the grid has a computational capac-

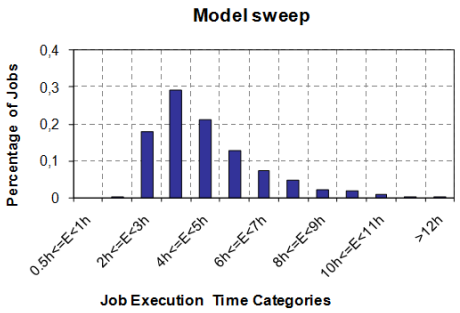


Figure 6.5: Task length distribution for model sweep-based jobs.

ity between 0.5 and 4 MIPS (uniformly distributed among CRs) and is limited to run a maximum of 2 tasks simultaneously. To focus on the variation of the execution time progress and on the accuracy of the predictions, we assume that once initialized, the computational speed of CRs remains unmodified during the whole simulation experiment.

The WAN links connecting the 4 distributed sites transfer data at a constant rate of 5 Mbit/s, with a latency uniformly distributed between 3 en 10 ms per link. On the other hand, intra-site transfers occur with a maximum speed of 1 Gbit/s. However, since link capacity within LANs is shared among the active data transfers, the more data traffic has to be processed, the lower the transfer rate. Finally, all the LAN links possess a fixed latency of 1 ms per link within a transfer route.

As was mentioned earlier, job parameters for the simulated grid model are derived from an existing tool for modeling and virtual experimentation with environmental systems, called Tornado. Tornado possesses a broad category of jobs, or *virtual experiments*, with input dependencies. In this work, we consider a group of input dependencies resulting from *model sweeps*. It means that tasks are derived from different mathematical models and have strongly varying execution times. The considered execution times (E^{act}) on CR^{ref} and their variations are depicted in Figure 6.5.

In this work, we simulate jobs containing 10 dependent tasks on average, organized into a 3-level dependency hierarchy. For simplicity, we assume that input, output, and checkpointing data of all tasks are equally sized and amount to 1 GB. A checkpointing delay of 2 s is also identical.

The task parameters considered correspond with the actual task properties observed when running Tornado experiments on the UGent grid infrastructure [15]. The UGent grid is a part of the Belgian grid infrastructure and consists of 76 CRs

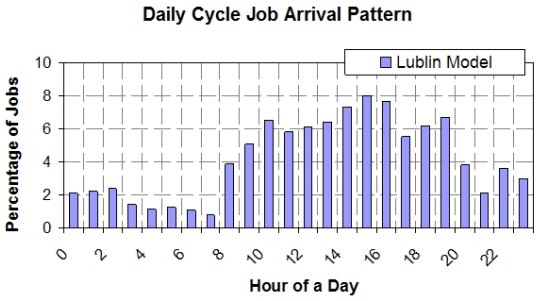


Figure 6.6: Job arrival pattern with daily cycle. U(U_{min}, U_{max}) = uniform distribution within U_{min} and U_{max}.

having a total of 222 CPUs, 304 GB memory and 4.4 TB disk space.

The workload described above is submitted into the considered grid environment according to the Lublin job generation model [16]. The model implies that most of the jobs (about 80%) arrive during day time, between 8 AM and 8 PM, resulting in peak hour loads alternating with relatively idle periods, as shown in Figure 6.6. This cyclic behaviour largely corresponds to the behaviour of Tornado users observed on the UGent grid.

We assume that an equal number of tasks following the *overestimate*, the *underestimate* and the *fluctuating* progress evolution models are submitted. The model parameters are initialized as follows (see also 6.1): E^{ref} for the *overestimate* model is uniformly distributed between $E^{act}\times150\%$ and $E^{act}\times200\%$; E^{ref} for the *underestimate* model equals $E^{act}\times10\%$ + A; E^{ref} for the *fluctuating* model equals E^{act}; A is uniformly distributed between $E^{act}\times150\%$ and $E^{act}\times200\%$; b is uniformly distributed between 0 and 1; F is uniformly distributed between $E^{act}\times5\%$ and $E^{act}\times10\%$; φ is uniformly distributed between $E^{act}\times1\%$ and $E^{act}\times2\%$; and, finally, we observe 3 types of noise $r_2 = 0$ (no noise), $r_2 = 0.05A$ (low noise oscillation amplitude) and $r_2 = 0.5A$ (high noise oscillation amplitude). The task execution progress is updated every $E^{act}\times0.5\%$.

Finally, it is also important to mention that we utilized the Tornado modeling and virtual experimentation framework for fitting periodic extrapolated execution time predictions to a number of predefined functions within the prediction algorithm. Tornado routines were called from the DSiDE code using the TornadoCPP Software Development Kit (SDK) that consists of a Dynamically Linked Library (DLL), an import library and corresponding header files. In particular, to perform nonlinear curve-fitting (data-fitting) within Tornado, the following procedure is to be followed:

- Implementation of the prediction evolution functions as algebraic models,

Parameter	Value
A	U($E^{ref} \times 150\%$, $E^{ref} \times 200\%$)
$Activationintervalof IS$	5 min
b	U(0, 1)
B of LAN links	1 Gbit/s
B of WAN links	5 Mbit/s
C	2 s
C_J	1 GB
$CRNumber$	128
F	U($E^{ref} \times 5\%$, $E^{ref} \times 10\%$)
I_J	1 GB
$Latency$ of LAN links	1 ms
$Latency$ of WAN links	U(3 ms, 10 ms)
$MIPS_{CR}$	U(0.5 MIPS, 4 MIPS)
$MIPS_{CR^{ref}}$	1 MIPS
n_{CR}^{max}	2
$n_{CR^{ref}}^{max}$	1
O_J	1 GB
P	U(1%, 100%)
$SchedulingintervalofGS$	10 min
φ	U($E^{ref} \times 1\%$, $E^{ref} \times 2\%$)

Table 6.1: Listing of model parameters.

specified in one of the two modeling languages supported by Tornado: Model Specification Language (MSL) [17] or Modelica [18].

- Creation of a Simulation Experiment (ExpSimul) that simulates the models over the desired time interval.
- Creation of an Objective Evaluation Experiment (ExpObjEval) that calculates the Sum of Squared Errors (SSE) between the values simulated by ExpSimul and the input data.
- Creation of an Optimization Experiment (ExpOptim) that executes ExpObjEval iteratively for different model parameters, until the minimum SSE is reached. We have used the Simplex [19] optimization solver to provide initial model parameter values.

6.5.2 Simulation Results

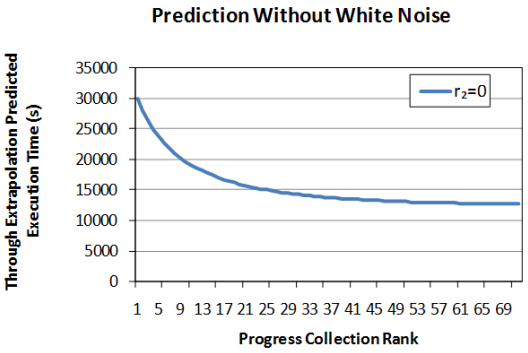
In this section we compare the performance of the dynamic algorithm when using two different task execution time prediction approaches. In the first approach the prediction value is simply derived by extrapolation from the last task progress measurement. The second approach is the proposed curve-fitting based prediction mechanism. The performance of both methods is observed for job progress curves with different amplitude white noise. An example for the *overestimate* model in Figure 6.7 suggests that we consider job execution time predictions with a perfectly exponential course (or a sinusoidal course in the case of the *fluctuating* model), as well as jobs with noise with low and high amplitudes.

The simulation results on the performance of the two prediction algorithms are depicted in Figures 6.8 and 6.9. In concreto, Figure 6.8 (a) shows the fraction of *useful workload* processed by the dynamic scheduler in both cases. The term *useful workload* refers to the total processing time on CR^{ref}, spent running successfully executed jobs. Formally this definition can be written as follows:

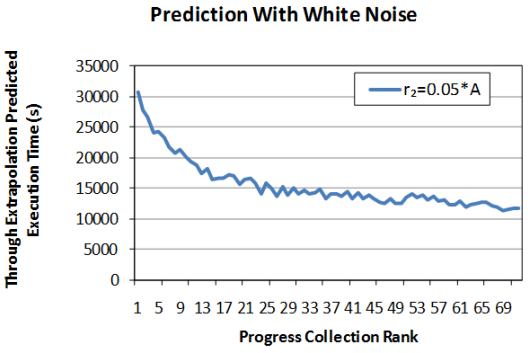
$$E = \frac{\sum_{J \in Done} E_J^{CR^{ref}}}{\sum_{J \in Submitted} E_J^{CR^{ref}}} \tag{6.6}$$

where $Done$ is a set of jobs, for which the final result is successfully computed within the observed time interval; and $Submitted$ is a set of all submitted jobs. We have to emphasize that successfully executed tasks, belonging to jobs that have not managed to execute within the predefined interval, do not contribute to the *useful workload*.

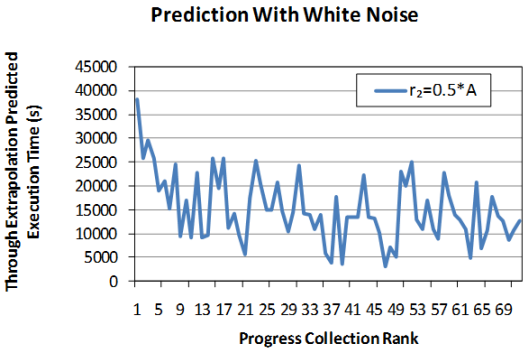
The simulation results suggest that when the curve-fitting-based prediction method is used, the dynamic algorithm achieves up to 15% better performance, compared to the case when the extrapolation method is applied. The advantage of the curve-fitting procedure is particularly remarkable in the case of noise with low amplitude ("Low Noise"). In this case the trend of the progress curve is preserved,



(a)

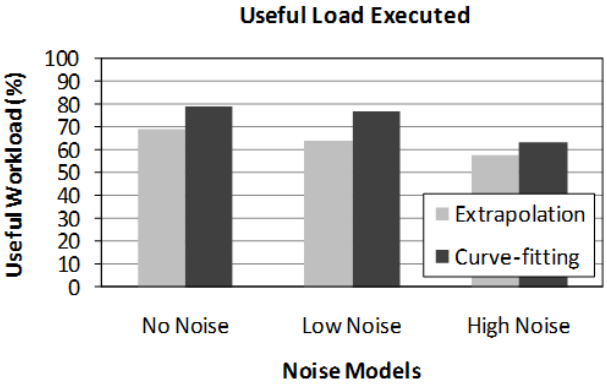


(b)

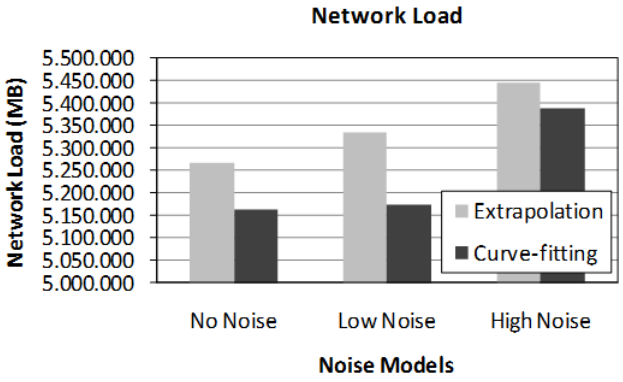


(c)

Figure 6.7: Examples of white noise, used to evaluate the prediction algorithm performance (a) exponential progress evolution without noise, (b) exponential progress evolution with low amplitude noise and (c) exponential progress evolution with high amplitude noise.

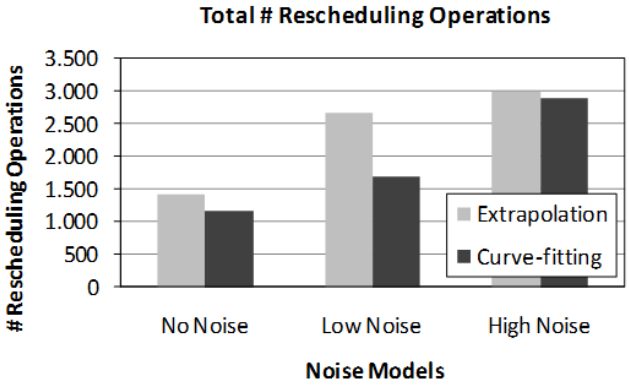


(a)

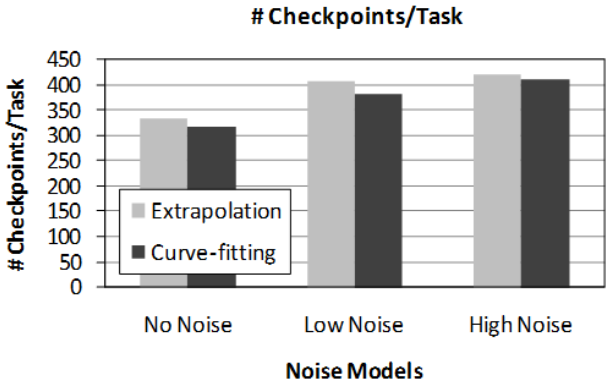


(b)

Figure 6.8: Performance comparison between extrapolation-based and curve-fitting-based prediction approaches: (a) proportion of successfully processed *useful workload*; (b) network load.



(a)



(b)

Figure 6.9: Performance comparison between extrapolation-based and curve-fitting-based prediction approaches: (a) total number of migrations within the time interval observed; (b) number of checkpoints per task.

simplifying the selection of an appropriate *prediction evolution* function and thus giving a correct indication of the total execution time in an early stage of a task processing. It means that the task can be assigned to the best suited resource at the beginning of its execution. Furthermore, matchmaking with an exponential curve largely eliminates oscillations in prediction values, reducing overzealous checkpointing and migration. The latter statement is confirmed by the simulation results in Figures 6.8 (b) and 6.9 (a) – (b), which show respectively the proportions of rescheduling operations, network traffic and checkpoints performed by the curve-fitting and the extrapolation-based algorithms.

Obviously, when the prediction curves show smooth evolution ("No Noise"), extrapolation-derived values show less variation and thus need less rescheduling operations to calibrate task execution times. Therefore, the performance of the dynamic algorithm with extrapolation gets closer to the performance of the curve-fitting-based algorithm, compared to the "Low Noise" case.

Finally, large amplitude white noise ("High Noise") conceal the trend of the prediction curve, obstructing the curve matchmaking procedure. The curve-fitting algorithm takes in this case wrong fitting decisions regularly, by selecting an inappropriate *prediction evolution* function to match with. In Figures 6.9 (a) – (b) can be observed that the numbers of migrations and checkpoints performed by both algorithm get close to each other. As a result, the curve-fitting approach performs only slightly better than the extrapolation-based method with respect to the *useful workload* processed.

6.6 Conclusion

In large distributed environments it is difficult to determine the execution time of applications due to a variety of input parameters, internal application dynamics and changing properties of distributed resources. However, knowledge of the total job execution time is essential for the implementation of an efficient scheduling policy. As this issue is difficult to address in a generic way, we consider a group of applications for which the execution progress can be monitored at run-time. An on-line prediction approach is proposed that uses the progress history to determine the course of the prediction curve and thus to estimate the total execution time. To achieve this goal, the approach makes use of curve-fitting of the current prediction evolution data to a number of predefined models.

To evaluate the prediction approach performance, the latter is integrated into an existing dynamic scheduler for workflow applications. The scheduler is in turn implemented in a grid simulator, called DSiDE, where a realistic medium-sized distributed environment with computational load derived from a real-world biological application is simulated. Under these conditions, the performance of the dynamic scheduler was evaluated for two situations: the scheduler uses the pro-

posed prediction mechanism; the scheduler uses an extrapolation-based execution time predictor. The simulation results suggest the performance improvement of up to 15% in the former case.

References

[1] M. Chtepen, F.H.A. Claeys, B. Dhoedt, F. De Turck, P. Demeester, and P.A. Vanrolleghem. *Scheduling of Dynamic Workflows in Grids*. IEEE Transactions on Parallel and Distributed Systems (In Review), 2010.

[2] M. Chtepen, F.H.A. Claeys, B. Dhoedt, F. De Turck, P.A. Vanrolleghem, and P. Demeester. *Dynamic Scheduling of Computationally Intensive Applications on Unreliable Infrastructures*. In Proceedings of the 2nd European Modeling and Simulation Symposium (EMSS '06), Barcelona, Spain, October 4 – 6 2006.

[3] F. Claeys, D.J.W. De Pauw, L. Benedetti, I. Nopens, and P.A. Vanrolleghem. *Tornado: a Versatile and Efficient Modelling & Virtual Experimentation Kernel for Water Quality Systems Applications*. In Proceedings of the International Environmental Modelling and Software Conference (iEMSs), Burlington, Vermont, USA, July 9–13 2006.

[4] A. Mandal, K. Kennedy, C. Koelbel, Marin G., J. Mellor-Crummey, B. Liu, and L. Johnsson. *Scheduling Strategies for Mapping Application Workflows onto the Grid*. In Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing, pages 125 – 134, Research Triangle Park, NC, USA, July 24 – 27 2005.

[5] M. Iverson, F. Ozguner, and L. Potter. *Statistical Prediction of Task Execution Times Through Analytic Benchmarking for Scheduling in a Heterogeneous Environment*. IEEE Transactions on Computers, 48(12):1374 – 1379, December 1999.

[6] C.-Z. Xu, L.Y. Wang, and N.-T. Fong. *Stochastic Prediction of Execution Time for Dynamic Bulk Synchronous Computations*. The Journal of Supercomputing, 21(1):91 – 103, January 2002.

[7] S.M. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X. Collazo. *A Modeling Approach for Estimating Execution Time of Long-running Scientific Applications*. In Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), the Fifth High-Performance Grid Computing Workshop (HPGC-2008), Miami, FL, USA, April 14 – 18 2008.

[8] B.J. Lafreniere and A.C. Sodan. *ScoPred – Scalable User-Directed Performance Prediction Using Complexity Modeling and Historical Data*. In Proceedings of the 11th Workshop on Job Scheduling Strategies for Parallel Processing, pages 62 – 90, Cambridge, MA, June 19 2005.

[9] B.-D. Lee and J.M. Schopf. *Run-Time Prediction of Parallel Applications on Shared Environments*. In Proceedings of the 5th IEEE International Conference on Cluster Computing (CLUSTER'03), Hong Kong, December 01 – 04 2003.

[10] M. Iverson, F. Ozguner, and G.J. Follen. *Run-Time Statistical Estimation of Task Execution Times for Heterogeneous Distributed Computing*. In Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing (HPDC-5 '96), Syracuse, New York, August 6 – 9 1996.

[11] F. Nadeem and T. Fahringer. *Using Templates to Predict Execution Time of Scientific Workflow Applications in the Grid*. In Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pages 316 – 323, Shanghai, China, May 18 – 21 2009.

[12] L.N. Nassif, J.M. Nogueira, M. Ahmed, A. Karmouch, R. Impey, and F. Vinicius de Andrade. *Job Completion Prediction in Grid Using Distributed Case-based Reasoning*. In Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WET-ICE), Linkoping, Sweden, June 13 – 15 2005.

[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. *High-performance, Portable Implementation of the MPI Message Passing Interface Standard*. Parallel Computing, 22(6):789 – 828, 1996.

[14] H. Casanova and L. Marchal. *A Network Model for Simulation of Grid Application*. Technical report, École Normale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme, 2002.

[15] UGent. *BeGrid UGent*. http://begrid.atlantis.ugent.be/.

[16] U. Lublin and D.G Feitelson. *The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs*. Parallel & Distributed Computing, 63(11):1105 – 1122, November 1992. 2003.

[17] H. Vanhooren, J. Meirlaen, Y. Amerlinck, F. Claeys, H. Vangheluwe, and P. Vanrolleghem. *WEST: Modelling Biological Wastewater Treatment*. Journal of Hydroinformatics, IWA Publishing, 5(1), 2003.

[18] Modelica consortium. *Modelica Website*. http://www.modelica.org/.

[19] W.H. Press, S.A. Teukolsky, W.T. Vettering, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992. Chapter 10, pp 408–412.

7 Conclusions and Perspectives

In this dissertation, two important aspects of application execution in distributed computational systems were investigated: fault-tolerance and scheduling of application workflows. This chapter highlights the most important contributions of this work and summarizes perspectives for future research.

7.1 DSiDE Simulator

A discrete event simulation environment, called DSiDE, was designed and implemented in the scope of this dissertation. DSiDE has a general, clear and extensible architecture, but thus far was mainly applied to performance evaluation of dynamic scheduling algorithms in grid-like systems. Therefore, the simulator principally contains built-in grid components, which however, can easily be completed with components for different types of distributed environments. For example, recently performance of thin client systems was tested in DSiDE.

The main advantage of DSiDE, compared to other existing grid and general purpose simulators, is that it allows for easy and flexible modeling of system and job dynamics. Examples of such events are resource failure and restart, changes in resource load, varying job progress evolution *etc*.

One of the main disadvantages of DSiDE is that it currently supports only modeling of centralized, *i.e.* bottleneck sensitive, distributed environments with a single scheduler and a single information service. This limitation should in the future be eliminated by providing a means for hierarchical or decentralized dis-

tributed systems modeling. Another practical improvement to DSiDE would be to allow addition of new computation and data scheduling algorithms in the form of Dynamically Linked Libraries (DLLs), which would avoid the necessity of system rebuilds when scheduling strategies are added or modified.

7.2 Fault-Tolerance in Distributed Systems

Adaptive replication- and checkpointing-based algorithms were introduced to overcome shortcomings of existing static fault-tolerant solutions. The main advantage of the proposed methods is that they modify the number of job replicas and the checkpointing interval at run-time as a function of system load and the resource failure frequency observed. Furthermore, to take advantage from the fact that resources within large distributed environments often alternate between heavy system load and idle periods, a hybrid scheduling algorithm was developed. The algorithm switches between checkpointing and replication dynamically, depending on system load monitored. The results have shown that dynamic selection of checkpointing intervals significantly reduces the run-time overhead in comparison with periodic checkpointing. On the other hand, adaptive replication-based solutions can provide fault-tolerance at lower cost in systems with low and variable load, by postponing replication in function of system parameters. Finally, the advantages of both techniques are combined in the hybrid approach that can best be applied when the distributed system properties are not known in advance.

A possible extension to the replication-based algorithms proposed would be to take into account the data transfer overhead, when deciding on replica location. Considering checkpointing-based algorithms, more refined criteria for checkpointing interval determination can be taken into account. For instance, profiles can be constructed for each resource or group of resources, based on system logs, to determine failure and restore patterns more precisely.

7.3 Scheduling of Workflows in Distributed Systems

A novel dynamic approach for balanced execution of workflow application was proposed in this work. The approach deals with applications with loosely coupled input dependencies, for which execution progress can be monitored at run-time. Based on this progress information, the algorithm makes periodic predictions on remaining execution times of tasks within each workflow. Tasks are scheduled in such a way that the makespan of each workflow is minimized, by balancing execution times of parent tasks, generating input for the same group of dependents. This procedure ensures that short non-critical tasks (*i.e.* outputs of these tasks can not be used immediately by dependent tasks) are scheduled on slow resources, while

compute intensive tasks get faster processing. The novelty of the algorithm lies in the combination of its balanced approach with a dynamic, progress information driven, rescheduling phase. Simulation results suggested a significant reduction (up to 35%) of job makespan when the dynamic balanced algorithm is applied, compared to the makespans achieved by a static scheduling solution.

Our research has considered static workflows, which means that the workflow structure is not subject to changes during run-time. As some applications contain dynamic task dependencies, it would be appropriate to extend this work to these more complex type of workflows.

7.4 Execution Time Prediction

An online task execution time prediction approach, which makes use of historical information on task progress evolution, was introduced. The approach matches task progress evolution curves against a number of predefined models by means of a nonlinear curve-fitting procedure. The prediction method proposed is general in nature and can be used as a plug-in for existing scheduling algorithms.

While most of the prediction mechanisms described in literature are complex and time consuming (as they either rely on detailed modeling of application components or on an extended set of historical performance records) the main advantages of the algorithm proposed are its simplicity and the ability to make relatively accurate predictions even with a limited number of historical data records available. The algorithm was integrated into the dynamic algorithm for workflows proposed earlier in this dissertation, enhancing its performance with up to 15%, compared to the case when extrapolation-based execution time predictions are utilized.

Currently, prediction evolution models used by the algorithm should be specified manually by end-users, based on their knowledge of the applications being executed. In practice, it would be more convenient to extend the model base dynamically, as more application types with different progress evolution curves arrive into a distributed system. This dynamic approach will require in the first place an unambiguous model classification procedure. Furthermore, the prediction algorithm proposed leaves space for further optimizations of the workflow scheduling algorithm introduced earlier. For example, the uncertainty of predictions made (determined by means of non-linear regression) can be used to refine the task rescheduling procedure within the algorithm: high uncertainty, in combination with low computational gain, can suggest that migrations should be omitted.